

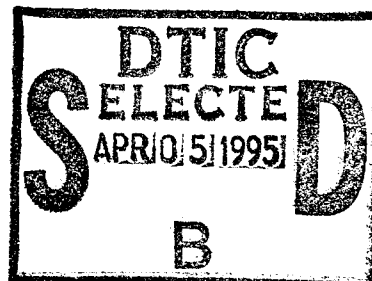
RL-TR-94-237
Final Technical Report
December 1994



INTEGRATED HIGH PERFORMANCE DISTRIBUTED SYSTEM

BBN Systems and Technologies

Edward F. Walker, Christopher Barber, Buz Owen, John Zinky,
Carl Howe, and Linsey O'Brien



APPROVED FOR PUBLIC RELEASE; DISTRIBUTION UNLIMITED.

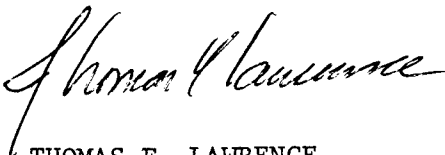
19950403 030

Rome Laboratory
Air Force Materiel Command
Griffiss Air Force Base, New York

This report has been reviewed by the Rome Laboratory Public Affairs Office (PA) and is releasable to the National Technical Information Service (NTIS). At NTIS it will be releasable to the general public, including foreign nations.

RL-TR-94-237 has been reviewed and is approved for publication.

APPROVED:



THOMAS F. LAWRENCE
Project Engineer

FOR THE COMMANDER:



HENRY J. BUSH
Acting Deputy Director
Command, Control & Communications Directorate

If your address has changed or if you wish to be removed from the Rome Laboratory mailing list, or if the addressee is no longer employed by your organization, please notify RL (C3AB) Griffiss AFB NY 13441. This will assist us in maintaining a current mailing list.

Do not return copies of this report unless contractual obligations or notices on a specific document require that it be returned.

REPORT DOCUMENTATION PAGE

Form Approved
OMB No. 0704-0188

Public reporting burden for this collection of information is estimated to average 1 hour per response, including the time for reviewing instructions, searching existing data sources, gathering and maintaining the data needed, and completing and reviewing the collection of information. Send comments regarding this burden estimate or any other aspect of this collection of information, including suggestions for reducing this burden, to Washington Headquarters Services, Directorate for Information Operations and Reports, 1215 Jefferson Davis Highway, Suite 1204, Arlington, VA 22202-4302, and to the Office of Management and Budget, Paperwork Reduction Project (0704-0188), Washington, DC 20503.

1. AGENCY USE ONLY (Leave Blank)		2. REPORT DATE December 1994		3. REPORT TYPE AND DATES COVERED Final Aug 92 - Feb 94	
4. TITLE AND SUBTITLE INTEGRATED HIGH PERFORMANCE DISTRIBUTED SYSTEM				5. FUNDING NUMBERS C - F30602-92-C-0102 PE - 62702F PR - 5581 TA - 21 WU - AF	
6. AUTHOR(S) Edward F. Walker, Christopher Barber, Buz Owen, John Zinky, Carl Howe, and Linsey O'Brien					
7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES) BBN Systems and Technologies 10 Moulton Street Cambridge MA 02138				8. PERFORMING ORGANIZATION REPORT NUMBER BBN Report No. 7987	
9. SPONSORING/MONITORING AGENCY NAME(S) AND ADDRESS(ES) Rome Laboratory (C3AB) 525 Brooks Rd Griffiss AFB NY 13441-4505				10. SPONSORING/MONITORING AGENCY REPORT NUMBER RL-TR-94-237	
11. SUPPLEMENTARY NOTES Rome Laboratory Project Engineer: Thomas F. Lawrence/C3AB/(315) 330-2805					
12a. DISTRIBUTION/AVAILABILITY STATEMENT Approved for public release; distribution unlimited.				12b. DISTRIBUTION CODE	
13. ABSTRACT (Maximum 200 words) The objective of the Integrated High Performance Distributed System (IHPDS) project is to design, implement, and demonstrate a distributed programming environment that will integrate and support very high bandwidth networks with heterogeneous computer architectures, including parallel and specialized processors, and that will support multiple programming models. The IHPDS project is developing the Photon distributed programming environment to meet the emerging needs of distributed application programmers. Photon provides an object-oriented programming model. This provides users with a consistent view of all services while insulating the user from the implementation details of those services. This enforcement of modularity and the separation of policy and mechanism make the object model a powerful way to build large-scale, distributed applications. Photon is a distributed computing environment that supports distributed application development across heterogeneous machine architectures and programming languages. It is particularly targeted to support the needs of high performance applications. Photon not only supports applications whose components are located within a single local area network, but also supports applications whose components are widely dispersed geographically.					
14. SUBJECT TERMS Distributed computing, High performance computing, Object-oriented systems, Distributed computing environment				15. NUMBER OF PAGES 56	
				16. PRICE CODE	
17. SECURITY CLASSIFICATION OF REPORT UNCLASSIFIED	18. SECURITY CLASSIFICATION OF THIS PAGE UNCLASSIFIED	19. SECURITY CLASSIFICATION OF ABSTRACT UNCLASSIFIED	20. LIMITATION OF ABSTRACT UL		

Table of Contents

Chapter 1. Introduction	1
1.1 Document Overview.....	2
Chapter 2. An Overview of Photon.....	3
2.1 Photon Mechanisms.....	3
2.2 Objects.....	4
2.3 Multiple Layers of Naming.....	4
2.4 Explicit Location Mechanism.....	5
2.5 Global Futures.....	5
2.6 Distributed Shared Memory.....	8
2.7 Language Veneers.....	9
2.8 Proxy Objects.....	10
Chapter 3. Photon Architecture.....	11
Section 3.1 The Basic Services Layer.....	12
3.1.1 Abstract Data Types.....	12
3.1.2 Messaging Support.....	12
3.1.3 Unique Ids.....	13
3.1.4 Communication Support.....	13
Section 3.2 The Advanced Services Layer.....	15
3.2.1 Object Registration.....	15
3.2.2 Object Implementation Support.....	15
3.2.3 Object Access.....	16
3.2.4 Global Futures.....	20
3.2.5 Distributed Shared Memory.....	21
Section 3.3 Photon Object Services Layer.....	23
3.3.1 Interface Definition Language.....	23
3.3.2 Language Veneers.....	23
3.3.3 Naming.....	27
3.3.4 Location/Binding.....	28
3.3.5 Photon ADT and Class Registration.....	28
Section 3.4 Inter-Layer Modules.....	30
3.4.1 Security.....	30
3.4.2 Monitoring.....	30
3.4.3 Administrative Domains	31
Chapter 4. Resource Management in the Photon System.....	32
4.1 Introduction.....	32
4.2 A Resource Management Perspective of a System	33
4.3 Overview of Resource Management Problems.....	35
4.3.1 Minimize Resources Problem.....	35
4.3.2 Adaptive Allocation Mechanism Problem.....	36

For	<input checked="" type="checkbox"/>
ad	<input type="checkbox"/>
ion	<input type="checkbox"/>

Availability Codes	
Dist	Avail and/or Special
A-1	

4.3.3 Regulate Usage Problem.....	37
4.4 Overview of the System Development Process.....	37

Chapter 5. Summary of Technical Progress and Suggestions for Further Research.....40

5.1 Technical Progress.....	40
5.2 Suggestions for Future Direction.....	42

Chapter 6. References.....43

Table of Figures

Figure 2-1: 25% savings in communication latency realized through use of global futures.....	7
Figure 2-2: 67% savings in communication latency realized through use of global futures.....	8
Figure 3-1: Photon Architecture.....	11
Figure 4-1: Resource management view of system characteristics.....	34
Figure 4-2: In order to make resource management problems tractable, most system characteristics are assumed constant.....	35
Figure 4-3: Evolution of objects, resources, and applications.....	38
Figure 5-1: Photon Implementation.....	41

Chapter 1. Introduction

The objective of the Integrated High Performance Distributed System (IHPDS) project is to design, implement, and demonstrate a distributed programming environment that will integrate very high bandwidth networks with and support heterogeneous computer architectures including parallel and specialized processors, and that will support multiple programming models

The driving motivation behind this project is the desire to make the most of emerging high performance computing environments. The future computing environment will be composed of high speed computers and high bandwidth networks, but the networks will have a "high" latency. Advances in computer architecture have resulted in orders of magnitude of improvement in processor speed since less than a decade ago. Even greater improvements in processor speed can be expected in the future. Network bandwidth has also improved substantially, from 10 megabit/second Ethernet networks to 100 megabit/second FDDI networks to even higher bandwidth ATM networks.

However, network latency has not improved substantially. For example, polling latency in high bandwidth FDDI networks can cause the transmission delay of small messages to be longer in FDDI networks than in Ethernets. While modern networks are providing a wider "data highway," the "speed limit" has not increased. The speed of light imposes a fundamental limit on achievable latency. Improvements in processor speed have far surpassed possible improvements in network latency. Network latency, measured in terms of MIPS, has become much higher.

The challenge to application developers in such an environment is to make effective use of processing speed and network bandwidth, while at the same time avoiding the costs of network latency. The IHPDS project is developing the Photon distributed programming environment to meet the emerging needs of distributed application programmers. Photon is designed to avoid unnecessary communication and otherwise minimize the effects of latency in its own implementation, and provides mechanisms to application builders to help them minimize message traffic and thereby lessen the effects of latency.

Photon must be capable of accommodating a wide variety of computers, networks, and programming languages. The performance demands and diversity of missions in C² environments require a heterogeneous computing and communications environment. The heterogeneous environment consists not only of general-purpose computers, but also supercomputers, parallel processors, graphics renderers, digital signal processors, and other specialized platforms. Each platform provides specialized resources to the computing environment, but the applications using these distributed resources require that they operate as a unified and consistent whole. Photon provides a uniform programming model over a wide range of computing platforms to achieve this goal.

Photon provides an object-oriented programming model. This provides users with a consistent view of all services, while insulating the user from the implementation details of those services. This enforcement of modularity and the separation of policy and mechanism make the object model a powerful way to build large-scale, distributed applications.

Nonetheless, there are many applications that do not naturally fit into this model. Photon provides a distributed shared memory model as an alternative to the object-oriented model for such applications.

Photon must be scalable to accommodate large user and service populations. As military networking expands to provide near-universal connectivity, the range of services that users of the network

require will also expand. In particular, we expect that there will be increasing information flow across service boundaries to support C² operations, and it is critical that Photon support and not hinder that expansion. Therefore Photon is designed for efficient large-scale operation to accommodate this expansion.

Photon is intended to be as simple and easy to understand as possible for its users. The complexities needed to manage distributed operation and to achieve high performance will be largely hidden behind easy to use interfaces.

Photon addresses the high cost of network latency in all aspects of its design, and in the mechanisms it provides. Avoiding unnecessary communication is the key to high performance applications in the developing distributed computational environment. Photon will provide a natural environment for heterogeneous, high-performance, distributed programming and will support such diverse applications as C², multimedia conferencing, distributed databases, and distributed simulations.

1.1 Document Overview

This document is organized around a discussion of Photon. Chapter 2 gives an overview of Photon. This chapter can be skipped by readers already familiar with the basics of Photon. Chapter 3 presents a detailed description of the Photon architecture developed under the IHPDS project. Chapter 4 discusses resource management in Photon. Chapter 5 presents a summary of the technical progress made under the IHPDS project, and makes recommendations of areas for further research. Chapter 6 contains the list of references.

Chapter 2. An Overview of Photon

Photon is a distributed system architecture intended to provide an object-oriented distributed computing environment for the next generation of high performance distributed applications. The system provides the application developer with tools which facilitate the effective utilization of very high bandwidth communication, parallel and specialized computer architectures, and allow the use of multiple programming paradigms.

The construction of a distributed application can be simplified by using a distributed computing environment. A distributed computing environment offers tools that facilitate the construction of clients and servers. Typically the application builder provides a specification of a server's interface; the distributed computing environment generates stubs from the user's interface description for use in client programs, and also generates the framework for a server supporting the interface. The programmer only has to be concerned with application-specific code; the distributed computing environment takes care of the details of message packing and communication.

Often distributed applications need to run over a mix of computer architectures and operating systems. Application builders might desire to reuse software written in several different programming languages, without having to translate it. Therefore, it is desirable for a distributing computing environment to support applications that run on a heterogeneous mix of machines, operating systems, and have components written in different programming languages.

Photon is a distributed computing environment that supports distributed application development across heterogeneous machine architectures and programming languages. It is particularly targeted to support the needs of high performance applications. Photon not only supports applications whose components are located within a single local area network, but also supports applications whose components are widely dispersed geographically.

2.1 Photon Mechanisms

Photon provides high-performance operation and data transfer while retaining an object-oriented programming model. Because Photon is object-oriented, it provides features such as abstraction, encapsulation, and modular client/server interfaces. Furthermore, it includes abstractions to enhance performance such as support for parallelism, explicit data dependence, data streaming and redirection, and location binding. Finally, it provides these features in a modular and customizable fashion that facilitates implementation on a wide variety of platforms.

Subsequent sections describe the key mechanisms within Photon. These mechanisms are:

- Objects
- Multiple layers of naming
- Explicit location mechanism
- Global futures
- Distributed shared memory
- Language veneers
- Proxy objects

2.2 Objects

The fundamental building block in Photon is the *object*. Photon objects have the following characteristics:

Objects are instances of classes that define both the data representations and operations for manipulating the data for that object.

The implementation of objects is encapsulated and hidden from the user.

Communication with objects is accomplished through messages passed between the user and the object implementation.

Objects can reside on multiple hosts in the environment and operations can be requested of them from any host.

Objects may or may not have internal state and this state may or may not be persistent.

Our definition of objects is very general because we do not wish to limit the ways in which objects can be manipulated. A very high performance distributed system should have the flexibility to optimize how operations are performed without causing the behavior of those objects to change. For example, if the system recognizes that a program and an object it manipulates reside on the same machine, direct access to the object's data through shared memory can permit object operations that have performance comparable to a simple subroutine call. However, the system can only permit this type of optimization if it can still guarantee that the program performs the operation correctly and that the object's integrity remains intact. Therefore, Photon limits the amount of information that a program and an object are permitted to know about each other to that specified in a public interface description. This restriction provides maximum flexibility in the implementation of the object and permits Photon to dynamically select from multiple implementations in order to maximize performance.

Objects are implemented in Photon by servers. Servers are simply programs that represent one or more objects and have a published set of operations that can be performed on those objects. Servers are the mechanisms through which objects are realized in the system; if an object has no server, no operations can be performed upon it until a server is created for it. All Photon services are provided either by servers or by code compiled or linked into the user's program. Because the built-in services provided with Photon are also provided within the object model, new services added by the user are indistinguishable from those provided by the system itself. Therefore, Photon can be considered an extensible distributed system.

2.3 Multiple Layers of Naming

Photon includes multiple levels of naming. These include *symbolic* names, *location-independent* or *logical* names, and *location-dependent* or *physical* names.

Symbolic names provide methods of addressing services that can be understood easily by humans. Symbolic names are usually alphanumeric strings such as "MandelbrotComputeServer." Symbolic names are converted into other types of names by a name service. The name service is implemented as a collection of servers that can be accessed by any Photon client. The function of these servers is very simple: given a symbolic name, they return a location-independent name.

Location-independent or Logical Object IDs (LOIDs) are used to identify a service without being specific regarding its location. LOIDs are fixed-length structures and are not necessarily easy to interpret other than by machine. However, they do provide a unique identification for a service

throughout the Photon system and this identification can be either stored or used to access servers through the locate operation. Note that LOIDs can be resolved to more than one object or server. Protocol stacks supporting multicast addressing may wish to associate multicast addresses with LOIDs.

Location-dependent names or Physical Object IDs (POIDs) are names that specify the physical location of an object in Photon with respect to the underlying communications network. Use of POIDs allows the system to associate an object with a communications link or connection and thereby optimize the communication over that link. These names are bound to the underlying communications protocols the application wishes to use.

2.4 Explicit Location Mechanism

Photon requires that a process know the location-dependent name of an object before it can perform operations upon it. The location-dependent name is discovered by doing a locate operation on a location-independent name. This means that even when an object may be replicated on many nodes, the system will select only one of these objects at the time a client does a locate operation. The client will continue to use that instance of the object until there is a reason not to (such as an exception).

This method is a departure from the approach taken in a system previously developed at BBN, the Cronus [BERETS89] [SCHANTZ85] distributed computing environment. In Cronus, because the system performs location transparently as needed, programmers only need to deal with location-independent names. This approach was not felt to be sufficient for Photon. Although Photon does permit programmers to delegate the mapping to the system if they desire, the system supports explicit programmer control over the mapping. Having control is important for high performance applications which may want to specify exactly when location is performed in order to control the timing of the necessary overhead. Furthermore, when several possible mappings for a LOID exist, it may be important for an application to be able to explicitly choose one that best satisfies its performance goals.

2.5 Global Futures

Many techniques exist for allowing a single process to start and control several concurrently executing activities. These include multitasking, futures, and streams. Multitasking approaches include the concept of threads provided by Mach [Rashid 1986] and by POSIX threads [POSIX 1992]. In multitasking systems, each thread may be blocked waiting for a different remote request. When the request is finally satisfied, thread execution continues, guided by normal scheduling policy. Futures include the approach taken by Cronus [Walker 1990]. Rather than blocking RPC requests until the value is returned, the request returns a "future" which can be claimed when the value is needed. A thread may have several pending futures, and may invoke requests that create more. Futures may be combined into a "future set" which can then be used to block the thread until one of the set's values has been received. Streams, such as those implemented in Mercury [Liskov 1988], allow a client to submit a sequence of requests and then claim the replies in the same order as the requests were given. This allows client and server to execute in parallel. This range of approaches is generally adequate for distributed processing driven by point-to-point requests between a single client and many servers.

The global future is an addition to the basic RPC/call stream model. In the global future model, a caller may issue remote procedure calls to various servers and receive the results of these calls at a later time. The asynchrony inherent in the future mechanism allows the client to issue many calls

before receiving the results of the first; this allows computation in the server to proceed in parallel with computation in the client, since the client does not have to wait for results from one call before issuing a second call.

Global futures extend this capability by allowing the client to issue calls to several servers, with output from a call executed by one server being transmitted directly to a second server. This forwarding of data allows computation in the server to proceed without the client being a bottleneck between them. The computation in the servers may also proceed in parallel.

A future is a typed value. It has a globally unique ID. When the client directs the server to perform a computation, it issues an RPC, for example: `ObjID.operation(input_param_list, output_param_list)`.

In a remote procedure call without global futures, only the values of the input parameters would be communicated to the server, and later the values of the output parameters would be sent back to the client by the server. Only then could the client supply the value received as an input parameter to another RPC call to a different server.

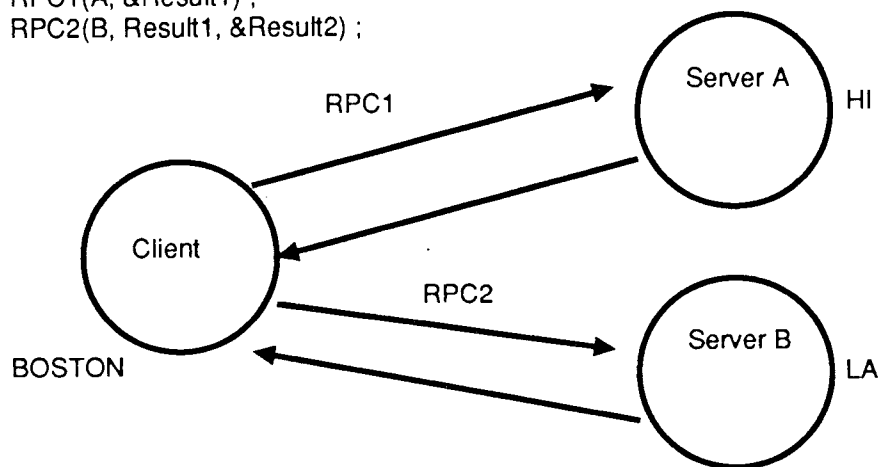
By using global futures, the client can cause a result to be forwarded directly from one server to another without its passing through the client. The client does this by specifying the ID of a global future in place of one or more of the input or output parameters. When a future is supplied instead of an output parameter, it will usually specify a distribution list. The distribution list directs the server to send a copy of the output value to each of the destination hosts in the list as soon as the value is computed. When an input parameter is specified as a global future, the server waits for the value to arrive, binds the value received to the parameter occupied by the future, and then proceeds with the operation requested. Since future IDs are allocated by the client without consulting the server, a client can set up a flow of values from one server to another with no waiting for intermediate results. As a result, the joint computation between the servers will complete as soon as possible.

Figures 2-1 and 2-2 illustrate how global futures can be used to reduce communication latency. In each example, a global future is used to eliminate one or more unnecessary message trips from the server to the client and back to the same server or a different one.

Client desires to call A, obtaining a result it then passes to B to obtain the final result.

Without Futures:

```
RPC1(A, &Result1) ;  
RPC2(B, Result1, &Result2) ;
```



With Futures:

```
RPC1(A, &futureResult1) ; (returns immediately)  
RPC2(B, futureResult1, &Result2) ;
```

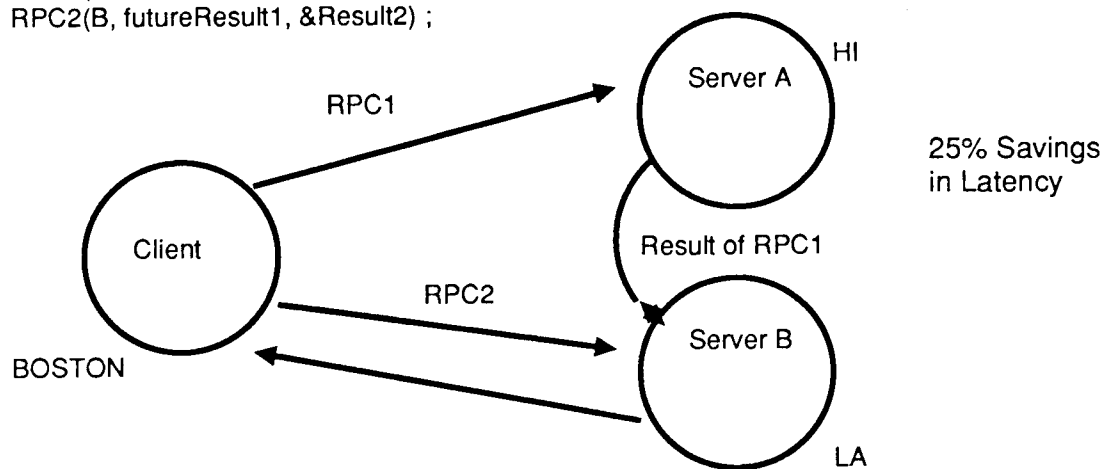
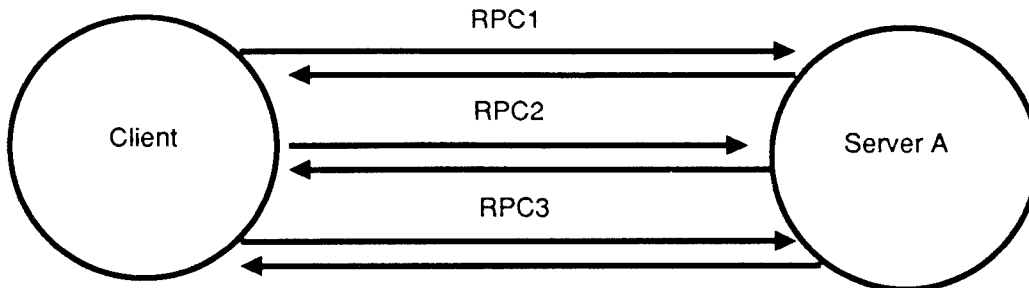


Figure 2-1: 25% savings in communication latency realized through use of global futures.

Without Futures:

```
RPC1(A, &Result1) ;  
RPC2(A, Result1, &Result2) ;  
RPC3(A, Result2, &Result3);
```



With Futures:

```
RPC1(A, &futureResult1) ;  
RPC2(A, futureResult1, &futureResult2);  
RPC3(A, futureResult2, &futureResult3);
```

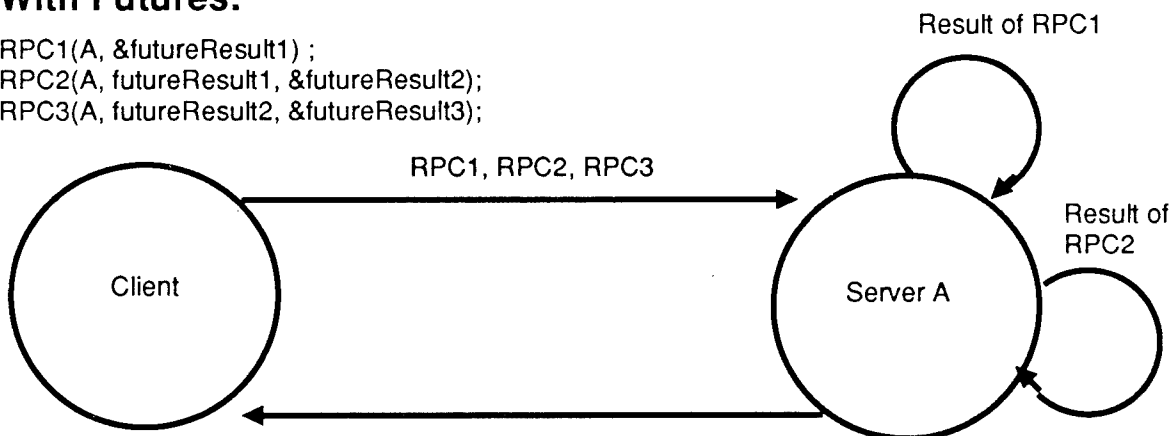


Figure 2-2: 67% savings in communication latency realized through use of global futures.

2.6 Distributed Shared Memory

Photon includes a distributed shared memory abstraction in addition to message passing based on Photon objects. Distributed shared memory is a more convenient abstraction than message passing for some applications.

Photon is targeted to a heterogeneous environment. This immediately raises two issues. First, there is the question of reconciling differing page sizes among heterogeneous platforms. Second, there is the question of differences in data alignment and representation between among such platforms.

Photon solves the problem of selecting a page size by not segmenting the distributed memory in terms of pages. Pages cause false sharing that can lead to unnecessary thrashing. A page can

contain several data objects; two processes that do not access any of the same data objects can end up trashing due to conflicts over pages that contain multiple objects that happen to be accessed by both processes.

Instead of segmenting memory by pages, Photon segments the distributed memory by abstract data object. Each segment of memory contains a complete abstract data object. The segments are therefore of variable size.

The solution to the second problem, that of handling different data representations, is related to the solution of the first. Each segment of memory contains an abstract data object whose type can readily be determined by Photon. Given this type, Photon can convert object instances of the type to and from a common external representation based on the specification of the type provided by the programmer in a special definition language (a slightly modified version of CORBA's IDL [CORBA93]). When a memory segment is mapped into a process' address space, Photon converts the enclosed data object from external format to local format. When a memory segment is propagated, Photon converts the enclosed data object to its external format.

Photon optimizes the case where two processes sharing the distributed shared memory reside on the same platform. In this case, actual shared memory is used.

To improve the performance of the distributed shared memory, Photon allows application programmers to provide information to the system about the intended pattern of memory use. This knowledge makes it possible for Photon to optimize performance and eliminate unnecessary communication. Programmers indicate an access pattern to Photon by tagging each memory segment with a memory access type. Several different memory access types are supported. Untagged memory segments are managed in a fashion that preserves consistency regardless of the pattern of use. This approach was first used in Munin [Bennett 1990].

Photon includes an important mechanism for reducing the latency of distributed shared memory access. In Photon, one object in a memory segment can contain a reference to an object in another memory segment. Photon permits application programmers to indicate that some of these references are "down pointers." By doing so, the programmer indicates to the system that the application is likely to access the referenced object if it references the enclosing object. Photon uses this information to prefetch segments that are the targets of down pointers when the referring object is accessed. If any of the prefetched segments are actually accessed, no time must be spent waiting for the segment to be propagated from a site with a copy.

Prefetching memory segments in this fashion trades off latency for network bandwidth. This is an appropriate trade off in the current state-of-the-art distributed environment. Gains in network bandwidth have far outpaced reduction of network latency, and this trend is sure to continue in the future, especially since improvements in latency are ultimately bounded by the speed of light while improvements in bandwidth do not face any such physical limit.

2.7 Language Veneers

One of the major goals of Photon is to provide a programming environment which is convenient and natural to its users, whatever programming language is being used. Since different programming languages such as C++, Lisp, and Ada encourage very different programming styles and paradigms, Photon tries to fit the style of the language being used as closely as possible. For this reason, rather than provide a single, monolithic API (Application Program Interface), the same in all languages, Photon provides a different language veneer for every supported language. Our initial efforts have been concentrated on constructing a language veneer for C++. We chose C++ to start with because that language's features allow us to hide many of the complex details of distributed programming and

present natural abstractions to the Photon user without having to extend the language. In the C++ veneer, a Photon object class is readily modeled as a C++ class.

Each language veneer has two main components: (1) an API for that language providing access to common Photon services, and (2) code generators which will produce declarations and code templates for object servers and the like.

2.8 Proxy Objects

One of the key elements in object-oriented language veneers is that of proxy objects. Proxy objects are class definitions that represent remote objects within a client program. These class definitions are created by the class designer to work cooperatively with servers of that object class. Proxy objects permit the class designer to implement arbitrary caching and consistency policies between clients and servers.

Proxy objects can play a key role in obtaining acceptable performance. Without proxy objects, a designer of an object interface can encounter a conflict between aesthetics and communication efficiency. If invoking each method necessarily entails a remote procedure call, the object designer will be tempted to trade off ease of use and define unnatural methods that try to piggyback several "natural" methods into a single call. With proxy objects, a method invocation on a Photon object does not necessarily entail any communication. The object designer can create an aesthetic object interface, and use a proxy object to obtain an efficient implementation.

Remote procedure calls are simply a special case of proxy objects. RPC stubs generated from an ODL are merely proxy objects that send a message to the server for each operation requested upon an object. However, it is not difficult to visualize object definitions where these trivial proxy objects cause unneeded communication. For example, imagine that there exists a class ball that has operations `define_color` and `read_color`. A client might contain code such as the following:

```
color c;  
ball b;  
b.define_color(red);  
...  
c = b.read_color();
```

Using an RPC proxy object, these two operations would require two separate messages to be sent to the ball server, and two replies to be received. However, a more intelligent proxy object might simply notify the ball server that it was maintaining a cache of b's object attributes and cache the color of the ball in a local state variable. Then when the `read_color` operation was requested, the local proxy object could reply red without any further communication with the server.

Chapter 3. Photon Architecture

This chapter is organized around a discussion of the Photon architecture illustrated in Figure 3-1. Each layer of the architecture is discussed in a subsequent section. The architecture has been designed to provide applications with high performance in an environment where communication has a high latency relative to processor speed. The architecture is divided into three layers: the Object Services layer, the Advanced Services layer, and the Basic Services layer. In the Photon architecture, a layer is permitted to use the functionality of any lower layer, but not an upper layer. Two functional modules do not fit into any particular layer, and straddle all the layers: security and monitoring.

Application Layer

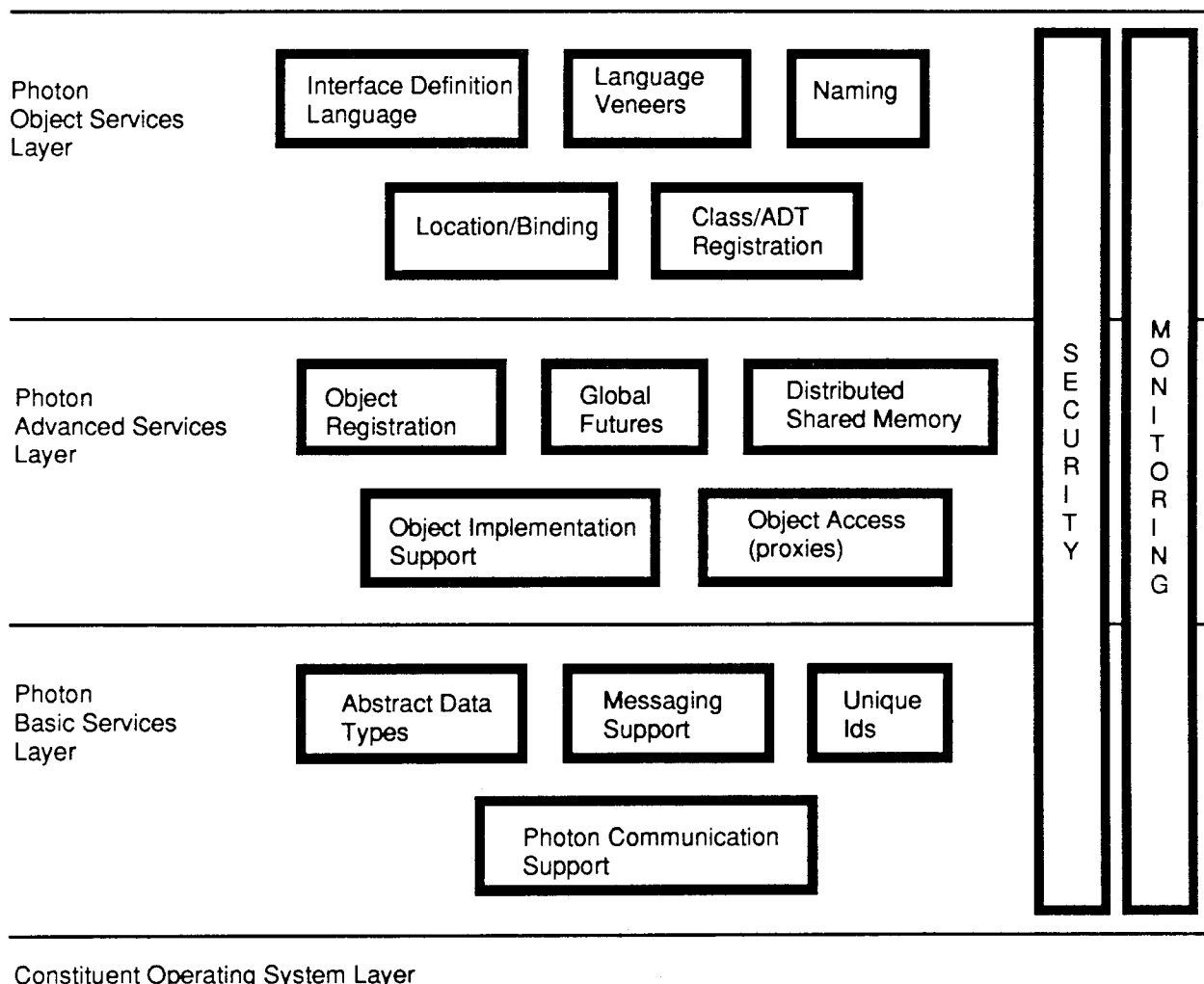


Figure 3-1: Photon Architecture

Section 3.1 The Basic Services Layer

The Basic Services layer is the lowest layer in the architecture and provides the basic functionality of Photon. This layer contains the following functional modules: Abstract Data Type Module, Messaging Support Module, Unique Id Module, and the Communication Support Module. (See Figure 3-1.)

3.1.1 Abstract Data Types

In a distributed program operating in a heterogeneous environment, it is necessary to have safe means to transmit data between sites with differing data-structuring schemes. Since the data needs to be converted from one format to the other, it is necessary for the conversion functions involved to know the structure of the data. This is best done by encapsulating such data within abstract data types which have associated conversion functions. In Cronus, this is done through "cantypes" which belong to a particular Cronus type (analogous to a Photon class); in Photon, ADTs (Abstract Data Types) provides such a mechanism. For a datum to be transmissible in Photon, it must be declared as an Abstract Data Type.

Photon ADTs differ from cantypes in a number of ways. Like cantypes, ADTs support basic data types such as integers and character strings and structured types such as records and arrays. ADTs also support tagged unions which are not currently supported for Cronus cantypes. Photon ADTs are grouped into packages in order to limit name clashing problems, but are not associated with any particular Photon class. A class need only import an ADT package in order to make use of the ADTs contained in it; a Cronus type, on the other hand, in order to use a cantype must be a descendant of the type which defines that cantype. ADTs have automatically generated encoding methods to convert data to and from a universal transmissible representation, have computationally inexpensive "as is" encoding methods for transmission between data-compatible architectures, and also allow programmers to create their own custom encoding methods for their ADTs.

Although Photon includes a number of built-in ADTs, most complex applications will require the definition of additional data types. Photon will provide an ADT definition language which will allow programmers to define packages of new ADTs. These specifications will be parsed and converted to an internal form which may either be stored in an ADT database or in separate files. The internal form, called an ADT package descriptor, will act as a specification for code generators to create appropriate interface and implementation code for various language veneers, to which the programmer may add code implementing custom encoding methods.

Photon uses the CORBA Interface Definition Language (IDL) as the ADT definition language. The CORBA IDL supports the basic and structured types needed in Photon, and even has a public domain compiler.

3.1.2 Messaging Support

The Messaging Support module supports message assembly and buffering. The primary goal in the implementation of this module is to avoid unnecessary copying, which can significantly add to overhead. Various researchers have shown that unnecessary message copying and scanning can overwhelm the actual costs of necessary communication protocol processing [CLARK89, CLARK90, and THEKKATH93]. Some interesting research has also been done by [DRUSCHEL93] in protocol architectures for high-performance communication.

3.1.3 Unique Ids

The Unique Id module supports the generation of globally unique identifiers. These identifiers are used to provide unique type and class tags, and unique Photon object identifiers. These identifiers are unique across all threads on all possible Photon hosts. It is possible to safely and efficiently generate multiple unique identifiers (UIDs) in parallel, avoiding any sort of UID generation bottleneck due to having to communicate with some central service or having to block on a semaphore.

Unique identifiers (UIDs) have the following components:

- A globally unique host identifier**, indicating the host upon which the UID is generated. The host identifier is assigned to the host when Photon is installed, and is independent of the host's network address (although it might contain the network address). (This avoids many difficulties when a host's network address is changed.)
- A timestamp**, identifying the creation time of the UID. The timestamp may be in a system-specific format but should have a granularity at least small enough to guarantee that process and thread identifiers cannot be recycled nor all the values of the counter be run through during a single tick. A one second granularity should be sufficient for this purpose.
- A locally unique process**, and if necessary, thread identifier, specific to the host upon which the UID is generated. This uniquely identifies the thread of execution in which the UID was generated. Note that it is possible for a system to generate the same process or thread id for different threads over the course of time, but not within the granularity of the timestamp. (All systems we are aware of go through the entire range of id values before recycling one, thus in actual practice an id won't be used until a number of hours have gone by.)
- A counter**, specific to each thread generating a UID. This will allow the same thread to generate multiple UIDs within the granularity of a single timestamp. For instance, if the granularity of timestamps is 1 second, a thread may generate multiple UIDs in the same second by incrementing the counter for every new UID. The counter will roll over back to zero when it reaches its maximum value.

3.1.4 Communication Support

The Communication Support module provides the basic functionality needed to transmit and receive messages. It will support high performance communication over wide area and local area networks as well as local communication within a single host with one or more processors. Communication over a mix of networking technologies will be supported, including ATM, FDDI, and Ethernet.

Explicit object location, an IPC kernel-less architecture, and the use of a technique we call dynamic method binding permit achieving high performance communication in Photon. The Communication Support module will provide different implementations of communication tailored for various environments, principally, intra-host, LAN, and WAN communication. Photon uses dynamic binding at run time to make the choice of which communication implementation to use.

Dynamic method binding is invoked as part of the location of an object. When a client is bound to an object, the object engages in a negotiation with the Photon support code in the client to identify the best communication paths between the two processes. The object then instructs the client as to which compiled-in Photon routines to use to achieve this communication. The client code stores that information and uses it to select the optimal code for invoking the object methods. This technique allows the same program to be run under Photon in both distributed and parallel environments

without sacrificing performance in either. It also permits programs to be run in both LAN and WAN environments without sacrificing performance.

The intra-host communication implementation is used when method invoker and object can share physical memory. This implementation supports high performance for Photon applications running on a shared-memory multiprocessor. Sender and receiver will communicate by placing messages in a portion of memory mapped into both processes' address spaces. Although older Unix systems did not support shared memory, virtually all modern multi-user operating systems support this facility (e.g., VMS, SVR4, OSF/1, and Mach all support shared memory in some form). Thus, this approach should be feasible on all of the systems for which Photon will be supported, although the underlying implementations might be slightly different due to differences in the shared memory interfaces supplied by the constituent operating system.

High performance is the primary goal in the implementation of support for network communication. While the most straightforward implementation would be to build on top of TCP/IP, some researchers have abandoned TCP/IP and implemented directly on top of the network interface device in order to achieve the highest possible performance [THEKKATH93, KRST93]. This approach hinders portability as the system becomes dependent on operating system internals and network device controller particulars. Furthermore, implementing a protocol from scratch that will work in a wide area environment would mean reimplementing the flow control, packet fragmentation and reassembly, and other mechanisms of TCP/IP -- a formidable and duplicative task. Consequently, the communication implementations in the cited references choose to avoid these problems by restricting their domain to local area communication. Photon will take a similar approach: the WAN communication implementation will be built on top of TCP/IP while the LAN implementation will be built below TCP, in order to achieve better performance.

Section 3.2 The Advanced Services Layer

The Advanced Services layer is the intermediate layer in the Photon architecture and provides the functionality that supports Photon objects. This layer contains the following functional modules: Object Registration, Global Futures, Distributed Shared Memory, Object Implementation Support, and Object Access. (See Figure 3-1.)

3.2.1 Object Registration

The Object Registration service provides the means for Photon objects to publish their existence so that they can be located and accessed. Every Photon object has a unique object identifier; to publish itself, a Photon object will hand its identifier to the Object Registration service. The service will then be responsible for enabling the location of the object given its identifier.

Photon supports the notion of a wildcard object identifier. Instead of identifying a particular object instance, a wildcard identifier refers to any object of a specified class. Applications use wildcard identifiers in a situation where any object of a given class can be used satisfactorily. A common example of such a situation would be an object that performs computation, such as an object that performs matrix multiplication. An application which multiplies matrices probably does not need to specify which particular instance of the object performs the multiplication. In fact, forcing the application to make such a choice could lead to extra communication cost or poor load balancing if the application were to pick an object on a remote or overloaded machine. By binding to a wildcard identifier instead of an object identifier, the application allows Photon to make an appropriate choice of object instance based on distance and load. Since the class identifier will be an implicit part of each object's unique identifier, the Object Registration service will have the information needed to resolve wildcard identifiers, for any objects that publish themselves. The service will support two mappings for wildcard identifiers: one which transforms a wildcard identifier directly into a location and another which transforms a class identifier into a set of object identifiers, which can later be mapped into locations.

Photon supports plural object identifiers. A plural identifier encapsulates and hides the fact that a service may in fact be implemented by many server objects. In the case of a replicated object, for example, several object instances will each serve as a replicand. All the replicands collectively make up the service. In terms of correct operation, it does not matter which object instance replicand receives an application's method invocation (although it may matter in terms of performance). A plural identifier provides a name for the collection of replicands that make up a replicated object. By binding to a plural identifier, instead of an identifier of a specific replicand, the system can choose a replicand instance based on performance considerations. The replicated nature of the object can in fact be totally hidden from the application by the plural identifier. Object Registration will maintain the information about the collection of identifiers that make up a plural identifier. A plural identifier is allowed to contain other plural identifiers. Because of this, the Object Registration service will need to be able to detect and handle the expansion of plural identifiers with recursive definitions.

3.2.2 Object Implementation Support

The Object Implementation Support module provides the infrastructure that supports Photon objects. Application programmers need only provide the application-specific code necessary to implement an object's methods. Code supplied by this module will handle the tasks of decoding a newly received

method invocation message, unpacking the arguments, dispatching to the appropriate method routine provided by the application programmer, and packing up a reply message.

Photon objects will support multiple threads of control. Each method invocation will run in its own thread; multiple method invocations can run in parallel for the same Photon object barring any application-specific synchronization restrictions. The Object Implementation Support library will be thread safe, allowing preemptive threads to be used within a Photon object. Support for preemptive multithreading will be built on top of the POSIX thread interface (pthreads). Programmers using the Ada and C language veneers will use the pthreads interface directly, while those using C++ will be provided with easy-to-use classes which will hide some of the mundane initialization and management tasks but will still provide the same functionality as pthreads. This decision was motivated by the following considerations:

Functionality: Pthreads has sufficiently rich functionality for all potential Photon uses. It supports multiple thread-specific data keys, thread cancellation, user-specified cleanup routines, priority scheduling, and access to other scheduling attributes.

Familiarity: Pthreads is the emerging thread package standard. As such, its interface will become familiar to a wide audience of engineers engaged in multithreaded programming. It will be easier in the long run for Photon programmers to use the familiar pthreads interface than to have to learn a Photon-specific one.

Portability: Since it is a standard, pthreads will look the same on all compliant platforms, so programmers won't have to learn a new thread package for every new system. For this reason, we will avoid using any system-specific thread package directly (e.g., Solaris's thread package).

Economy of design effort: Much work has already gone into the design of POSIX threads. While it is likely that improvements could be made to the design, it is much more economical to put trust in the existing design rather than have to hash out the same problems over again in our own design.

Economy of implementation effort: Since it is the emerging standard, it is likely to be supported on most new platforms. It is the supported thread interface on OSF/1 and is promised by Sun for Solaris as soon as the standard becomes less volatile, although we have already written an implementation based on the Solaris-specific threads package.

3.2.3 Object Access

The Object Access module supports application access to Photon objects. Within the same address space, a Photon object may be accessed directly, otherwise, objects will be accessed through the medium of proxy objects. A proxy object is a local object that provides an interface to some remote object such that a client program may use the proxy just as it would the actual Photon object which it represents. Each proxy object belongs to a special proxy subclass of the Photon class which has the proxied object as a member. Although the proxy subclass will be generated automatically from the Photon class definition, the class designer will still have the opportunity to write special proxy methods and implement arbitrary caching and consistency policies between the proxy and the real object.

Proxy objects can play a key role in obtaining acceptable performance in a high latency environment. Without proxy objects, a designer of an object interface can encounter a conflict between aesthetics

and communication efficiency. If invoking each method necessarily entails a remote procedure call, the object designer will be tempted to trade off ease of use and define unnatural methods that try to piggyback several "natural" methods into a single call. With proxy objects, a method invocation on a Photon object does not necessarily entail any communication. The object designer can create an aesthetic object interface, and use a proxy object to obtain an efficient implementation.

An implementation technique similar to Photon's proxy objects has been applied in the new Windows NT operating system [C93]. Windows NT has an architecture based on the client-server model; applications run as clients of the operating system server. A performance problem with this approach is the cost of context switching. To get acceptable performance, Windows NT avoids contacting the server for every system call. System calls that do not require access to global data are handled in the application without contacting the operating system server. Data is also cached in applications so that some system calls can be processed without having to contact the operating system server. Photon provides proxy objects to avoid the latency of communicating with an object on every method invocation, exactly analogous to the way Windows NT avoids the latency of a context switch on every system call. The benefit of this approach is that a clean interface can be provided to the user without having to be concerned that each method invocation incurs a network latency cost.

The non-application-specific code making up proxy objects is automatically generated from a definition supplied by the user. This function is carried out by the Interface Definition Language module. Proxy objects for Photon system objects are provided in a library.

For some object implementations, the interface of the proxy object might differ from the interface of the actual underlying object. Such implementations have two interfaces, (1) the external interface presented to applications via the proxy object, and (2) the internal interface the Photon object presents to the proxy object. The internal interface (2) is hidden to application programmers. This might be convenient, for example, in a case where several method invocations on the proxy are piggybacked together in a single internal method invocation. The internal interface provides a method unlike any external method; the internal interface's method's signature accommodates passing all the arguments of the several piggybacked external methods at once.

3.2.3.1 Prefetching

As discussed in the Introduction, network bandwidth is dramatically improving, while small improvements in latency are being dwarfed by dramatic improvements in processor speed. In order for applications to achieve high performance in this environment, Photon needs to provide mechanisms that put the dramatically wider network bandwidth to effective use in order to ameliorate the high latency. One such mechanism is *prefetching*. Prefetching allows an application to guess what data will be required by its user in the future, and have that data move over the network to the application before it is needed. The application could guess wrong and request the transfer of unneeded data, but it will typically be faster to transfer data of questionable utility as long as some fraction of it can be used to immediately satisfy an application's request, rather than waiting until it is known exactly what is needed and incurring the cost of latency. Prefetching allows trading off the higher network bandwidth for decreased latency. Some of these issues are addressed in [TOUCH93].

Prefetching will only make sense in certain applications. If data is constantly being updated, then prefetched data might be invalid by the time the receiver tries to access it. The best use of prefetching is to retrieve data from a read-only data set, for example, a C² application might want to prefetch cartographic information.

Prefetching in Photon is done by calling a special method invocation stub specifically for prefetching in the application client. The special prefetching stub has only input arguments; output arguments are

omitted. The prefetching stub immediately returns. Since the application is just giving a hint to the system about what data might be needed in the future, no error indication is ever returned. Photon invokes the method and stores the result in a buffer. When a normal invocation of the same method with the same input argument values is made, the result stored in the buffer is used.

One issue in the implementation of prefetching is the management of the buffer used to store prefetched data. The buffer must be large enough so that useful data is not bumped out before it is needed. The buffer manager must adaptively measure how much data is being aged out of the buffer before it is used, and adjust the buffer size and age limit upward when apparently beneficial. Similarly, the buffer manager must adjust the buffer size downward when that appears appropriate.

Another issue in the implementation of prefetching is the provision for interrupting the transmission of large messages containing prefetched data. The transmission of normal method invocation request and reply messages takes priority over the transmission of prefetched data. If when transmitting a large message containing prefetched data, a normal message enters the queue to be transmitted, the transmission of the prefetched data should be interrupted, and the normal message transmitted. Then the transmission of the prefetched data should continue.

3.2.3.2 Object and Process Migration

Another approach to reducing latency is to minimize critical-path message traffic between remote hosts. Some gains can be had here through the use of globally claimable futures (described later). However, one situation with high overhead due to message traffic is where a client invokes many operations on a remote object which must be done consecutively because computation must be performed on the result of each operation before the next one can be invoked. In this case, globally claimable futures are not sufficient because of the computation performed between successive invocations. There are several ways to deal with this particular problem:

1. **Object replication**

Replicate the remote object on the local machine. This can be done by accessing the remote object through distributed shared memory. While some message traffic between the local and remote host will still be necessary to synchronize the state of the object, these need not be exchanged after every invocation. For instance, the client could lock the object, perform a hundred operations on the object, and unlock the object; only a couple of messages would need to be exchanged between local and remote hosts to synchronize the two versions of the object. Of course, in the meantime no other process would be able to access the object.

2. **Object migration**

Moving the object onto the local machine will also provide a savings in message traffic in this situation. Messages would only need to be used to transfer the state of the object to its new location. One advantage to this method over object replication as described above is that other processes would still be able to access the object while the client is using it. Of course, it would be necessary to provide a mechanism whereby the client could temporarily lock the object to the local host, in order to guarantee that the object would not be moved to some other location before the client is done with it.

3. **Process migration**

The above methods would not generally be appropriate when the remote object is tied to a particular host, such as objects providing a compute service or objects which provide an interface to a piece of hardware. In the case of a compute service, while it would be easy to replicate or move such an object, particularly if it had no state, doing so would defeat one of

the main purposes in having such a service, which is to distribute the computational load evenly across the available CPUs. In the case of a device interface object, the object could not be moved away from the host to which the device is attached and replicating the object would require very close synchronization and thus would not result in any savings in message traffic.

Since it is not feasible to move the remote object to the client host, the only solution is to move the client code to the object. This can be done within the existing framework by creating an object with a method which performs the invocations on the remote object and the intervening computations. This object may then be moved to the remote site and executed there instead of on the client host.

All of the above methods must address the problem of how objects' state and methods are to be distributed across multiple, heterogeneous hosts. Since object state will be in terms of Photon ADTs, transferring the state will only require the conversion of the ADTs to and from an appropriate transmissible form. The distribution of object methods, on the other hand, is a much thornier problem since code translation is generally much more expensive than data encoding. There are a number of approaches to the problem of moving object methods to remote hosts:

1. Preinstall compiled code

In this case, the code would have to be compiled and installed on all the hosts to which the object might ever need to be moved. This is efficient at runtime but requires foreknowledge of where the object might be moved. Since we would like the ability to move objects to a client host to reduce latency for some tasks, we would have to know all of the hosts on which clients could run in order to use this scheme. This would be overly restrictive on the application programmer and would require too much manual configuration of the user of the application. However, this scheme is probably the easiest to implement.

2. Move and compile code on the fly

The code for the objects methods could be transferred to the destination host and compiled on the fly. This method is merely an extension of the above one, since there is no reason that the code could not be compiled as part of the system configuration. The method is only practical if the code is known to be portable to the destination host environment and the host system is fast enough to compile the code in an acceptable amount of time. Even so, there is still likely to be a fairly large overhead involved in moving an objects methods and it this cost would have to be measured against the expected gain in terms of reduction of latency.

3. Write code in interpreted language

This scheme would allow us to forgo the cost of compilation, but the resulting code would probably not run as fast as comparable compiled code. Given our assumption that we will be more concerned about latency than CPU consumption as the technology advances, this seems a reasonable compromise. Of course, it would also be possible to provide a compiler for the language which can be used when it is known that the code will be used often enough to justify compiling it. Such compilation would not contribute to latency, since the code interpreter could be used while the compilation is running off-line. Although we would have to install an interpreter on every Photon host, this could be part of the regular Photon installation and would present no extra configuration work to the application developer or end-user.

4. Code translation

Finally, the compiled code of the object methods could be transferred to the remote host and translated to appropriate code for that host's environment. This approach requires no

interpreter but does require a code translator between every two different environments. A variant of this approach is to compile object methods to a universal low-level representation, which would require only one compiler and interpreter per environment. Either approach would require too much work for each new environment supported by Photon.

Initially in Photon, we will rely on preinstalled code, and later will experiment with on-the-fly compilation and possibly a simple interpreted language.

3.2.4 Global Futures

Many techniques exist for allowing a single process to start and control several concurrently executing activities. These include multitasking, futures, and streams. Multitasking approaches include the concept of threads such as that implemented in Mach [RASHID86]. In multitasking systems, each thread may be blocked while waiting for a different remote request; when the request is finally satisfied, thread execution continues, guided by normal scheduling policy. Futures include the approach taken by Cronus [WALKER90]. Rather than blocking RPC requests until the value is returned, the request returns a future which can be claimed when the value is needed. A thread may have several pending futures, and may invoke requests that create more. Futures may be combined into a future set which can then be used to block the thread until one of the set's values has been received. Streams, such as those implemented in Mercury [LISKOV88], allow a client to submit a sequence of requests and then claim the replies in the same order as the requests were given. This allows client and server to execute in parallel. This range of approaches is generally adequate for distributed processing driven by point-to-point requests between a single client and many servers.

The global future is an addition to the basic RPC/call stream model. In the plain future model, a caller may issue remote procedure calls to various servers and receive the results of these calls at a later time. The asynchrony inherent in the future mechanism allows the client to issue many calls before receiving the results of the first; this allows computation in the server to proceed in parallel with computation in the client since the client does not have to wait for results from one call before issuing a second call. Global futures extend this capability by allowing the client to issue calls to several servers, with output from a call executed by one server being transmitted directly to a second server. This forwarding of data allows computation in the server to proceed without the client being a bottleneck between them. The computation in the servers may also proceed in parallel.

A future is a typed value. It has a globally unique ID. When the client directs the server to perform a computation, it issues an RPC, for example:

```
ObjID.operation(input_param_list, output_param_list)
```

In a remote procedure call without global futures, only the values of the input parameters would be communicated to the server, and later the values of the output parameters would be sent back to the client by the server. Only then could the client supply the value received as an input parameter to another RPC call to a different server.

By using global futures, the client can cause a result to be forwarded directly from one server to another without its passing through the client. The client does this by specifying the ID of a global future in place of one or more of the input or output parameters. When a future is supplied instead of an output parameter, it will usually specify a distribution list. The distribution list directs the server to send a copy of the output value to each of the destination hosts in the list as soon as the value is computed. When an input parameter is specified as a global future, the server waits for the value to

arrive, binds the value received to the parameter occupied by the future, and then proceeds with the operation requested. Since future IDs are allocated by the client without consulting the server, a client can set up a flow of values from one server to another with no waiting for intermediate results. As a result, the joint computation between the servers will complete as soon as possible.

3.2.5 Distributed Shared Memory

Distributed shared memory (DSM) is an emerging alternative to message passing for constructing distributed applications. It is more convenient for many applications to be able to communicate through data in DSM than to have to rely on message passing alone. For example, applications written to run on multiprocessors usually make use a shared memory model. DSM makes porting these applications to a distributed environment much more straightforward than it would be without it [WILSON92].

Since one of Photon's primary goals is to support a heterogeneous environment, Photon's DSM will be shared by platforms with different architectures. There is a concern that DSM might not be able to perform adequately in such an environment. However, there is some good evidence that heterogeneous DSM can be competitive in performance with homogeneous DSM systems [ZHOU90].

Photon divides the DSM space into segments, each segment containing an abstract data type (ADT); this is similar to the approach taken in Munin [BENNETT90]. Each segment is called a memory object. A popular alternative is to divide the DSM into fixed-size pages. There are several problems with this approach. First, in a heterogeneous environment, machines commonly have different page sizes, and therefore there is no obvious natural choice of page size for the DSM. Second, the data in a page is not necessarily related. Two threads can end up having to share a page, even though they do not in fact share any data. This "false sharing" can lead to thrashing, which can severely impair an application's performance. The data in a Photon memory object is the representation of an instance of an abstract data type; only threads sharing the same ADT will share the memory object. Thrashing might still occur, but only because too many threads are in contention over the same datum.

As in Munin, each Photon memory object is assigned a memory type by the application. The memory type lets the application indicate to the system the style in which the memory object will be used, and therefore lets the system optimize its caching and update strategies for the memory object. One memory type is provided that will work for arbitrary patterns of access. Other memory types, adopted from Munin, are write-once (memory that is written when initialized but subsequently only read), write-many (memory that is written a lot by many different threads), result (write-many memory where the writes do not conflict), migratory (used by only a single thread at a time), producer-consumer (written by one thread, read by one or more others), and read-mostly (rarely written). The builders of Munin identified these memory types by studying actual applications, and then came up with a set of memory types that covered nearly all the observed cases, plus a general one that will work for all the rest.

Each memory object contains a single ADT. The ADT can be constructed out of several other ADT's, all of which reside in the memory object. The ADT module is used by the DSM to convert the ADT in a memory object into a suitable representation.

For synchronizing access to memory objects, the Photon DSM provides lock objects. The Photon DSM takes advantage of its knowledge of the holding and release of locks to optimize the propagation of updates. Photon uses the delayed update approach of Munin. This approach results in less communication than the popular approach of immediately invalidating all the copies of a memory object when some thread writes it, the approach taken in [ZHOU90], for example.

Photon DSM can either be temporary or persistent. Temporary DSM is used as a scratch pad for a computation being carried out by many threads in parallel, but is of no further use and vanishes once the threads complete. Persistent DSM survives beyond the lifetime of any given thread, and must be destroyed by explicit user action. Persistent DSM requires tolerance to machine and communication failures. Photon takes the approach that temporary DSM does not require tolerance to machine and communication failures, although these failures will be detected and the application notified that the DSM has failed. This allows the implementation of temporary DSM to have much less overhead than the implementation of persistent DSM.

One memory object can contain a reference to another memory object. To permit the system to do prefetching, application programmers can annotate that a particular reference in a memory object is a "down pointer." By doing so, the application indicates to the system that if a thread accesses the memory object, it is likely to follow the "down pointers." Therefore the system can reduce latency by prefetching the memory objects that are the targets of the down pointers before they are actually referenced. This approach trades off bandwidth for latency, an opportunistic approach in the Photon environment of high bandwidth but slow latency.

Section 3.3 Photon Object Services Layer

The Object Services layer is the top layer in the Photon architecture and provides the functionality upon which applications are written. This layer contains the following functional modules: Interface Definition Language, Language Veneers, Naming, Location/Binding, and Class/ADT Registration. (See Figure 3-1.)

3.3.1 Interface Definition Language

Distributed environments such as Cronus [WALKER90], DCE [SHIRLEY92], and CORBA commonly provide a language used by application programmers to describe the interface to a server. The server interface description is commonly used to generate RPC stubs and "boilerplate" code for the server. Likewise, Photon features a language to describe the interface of Photon objects, the Photon Interface Definition Language (IDL). The IDL is used to define classes. Classes are implemented in programs which act as servers for objects of the class, and are accessed by programs, known as clients. A program may be both a client and a server, even for the same type of object.

The designer of a Photon application designs one more classes by writing class descriptions in the Photon Interface Definition Language. A file containing this description is processed by the Photon IDL processor. It may be processed only for syntax checking, or, assuming it finds no errors, to create a class descriptor, which is itself a Photon object. Class descriptors may be stored in individual files or registered in a Photon Class database. Code generators for each language veneer will use the class descriptors as specifications from which to produce the necessary interface modules and templates of the implementation modules for that language.

The Photon IDL is based on the CORBA Interface Definition Language.

3.3.2 Language Veneers

The language veneer is a service of the Photon Object Services layer. It is integrated with the language processor being used, either in the form of additional libraries and definitions, or as a preprocessor for Photon programs. It draws upon the Photon Interface Definition Language and Class/ADT Registration modules at the same layer to permit use of previously defined classes and data definitions. There exists a separate language veneer for each language supported under Photon.

One of the major goals of the IHPDS project is to design a programming environment which is convenient and natural to its users, whatever programming language is being used. Since different programming languages such as C++, Lisp, and Ada encourage very different programming styles and paradigms, we would like for Photon to fit the style of the language being used as closely as possible. For this reason, rather than provide a single, monolithic API (Application Program Interface), the same in all languages, we will provide a different language veneer for every supported language. Eventually, we plan to support C, C++, and Ada. The C veneer could probably also be used from other unsupported languages, provided that the system linking conventions allow it. Our initial efforts were concentrated on a C++ language veneer. We chose C++ to start with because that language's features will allow us to hide many of the complex details of distributed programming and present natural abstractions to the Photon user without having to extend the language. For instance, the ability to define parameterized types using C++ templates provides a convenient means for implementing "smart" pointers to Photon objects which can hide binding details. The techniques

described below will be applicable to other language veneers to the degree that other languages provide features comparable or more powerful than those provided in C++.

Each language veneer has two main components: (1) an API for that language providing access to common Photon services, such as the generation of unique numbers (UIDs), object location and binding, etc.; and (2) code generators which produce declarations and code templates for user-defined Photon ADTs and classes. The language veneer will not preclude the Photon user from making use of non-Photon software packages and libraries.

In developing a C++ veneer we had a number of design goals (not all of which will be applicable to other languages). These are:

3.3.2.1 Avoid "namespace pollution"

The definition of many external global functions and variables often results in clashes with other software packages with similarly named entities. In C++, this problem can be partly alleviated by restricting external global symbols to class and class variable names, thus reducing the number of symbols which could clash. Global functions can be replaced with static member functions of a global class. For instance, a function returning the current version number of the local Photon installation would be defined as a static member function of the Photon class as opposed to a global function:

```
Photon::getVersion(&versionString) ; // static member function
::GetPhotonVersion(&versionString) ; // global function
```

Similarly, constants specific to a particular package within Photon can be made members of the class providing the interface to the package rather than defined externally:

```
class PhSomePackage {
public:
    enum { MAX_VALUE = 42; };
    ...
}
if (i < PhSomePackage::MAX_VALUE) ...
```

In order to lessen the chance that built-in Photon class names will clash with non-Photon class names, all classes provided by Photon begin with the prefix "Ph"; for instance, the Photon UID class is named PhUid. The class "Photon" is used to package static member functions and variables as shown above.

Note that other than through the use of preprocessor macros, which is a dangerous practice, there is no way to resolve a clash between identical class names without manually changing them. Since, we are in essence using C++ classes as packages, this problem is no different than that of duplicate package names in Ada or duplicate module names in Modula-3.

3.3.2.2 Hide Implementation details through data encapsulation

C++ allows us to declare some class members as private and therefore inaccessible outside the class (and its designated friends) and other members as public. We use this ability to "hide" the implementation details of Photon datatypes. For example, the users of Photon UIDs need not know the internal representation or composition of these datatypes, but only need to be able to generate them and compare them to one another.

By not allowing programmers to access data members directly, but instead only allowing access via the methods which are declared as public, the internal representation can be changed in a later version without requiring users to change any code, although recompilation would be necessary. For this reason, we avoid public data members in all built-in Photon datatypes.

3.3.2.3 Use "orthodox canonical form" whenever possible

In general we adhere to "orthodox canonical class form" [COPLIEN]. This form is needed for any class whose objects will need to be assigned or passed by value or whose destructor performs any significant work, and is generally recommended for all nontrivial classes. The form requires the explicit public definition of:

- o a default constructor (X::X())
- o a copy constructor (X::X(const &X))
- o an assignment operator (X& operator=(const &X))
- o a destructor (X::~X())

There may be some Photon classes that will need to deviate from this form, but only for an explicit reason. For instance, it might be desirable to prevent the user from instantiating an object without specifying an argument for the constructor, in which case the default constructor would be declared private instead of public.

3.3.2.4 Minimize need for programmer to explicitly code routine tasks

The goal is to make routine tasks, such as memory allocation and deallocation, ADT encoding and decoding, and message packing and unpacking, as painless as possible. C++ constructors and destructors are used to automatically allocate and deallocate memory needed by various Photon objects. Constructors are used to allow Photon ADTs to initialize themselves from their encoded representation. Operator overloading is used to present a natural interface to the user; for instance, Photon integer ADTs have the same operators defined for them as standard C++ integers.

Abstract base classes are used to specify standard interfaces to be inherited by Photon ADTs, Photon objects, and other Photon constructs. When invoking methods from a base class, it is not necessary to know the particular subclass of the objects. For example, the following code accepts an array of arbitrary ADTs and packs them in encoded form into a message:

```
PhMessage* packArray(PhAdt& adts[], int nadts)
{
    PhMessage *message = new PhMessage ; for (int i=0; i<nadts; i++)
        message->appendAdt(adts[i]) ; return message ;
}
```

C++ mechanisms may be used to define a "smart" pointer for object binding. Let us assume that we want to define a simple Photon class, named "Foo" with a single method "bar". The class specification would be done in a Photon-specific definition language, and after this is parsed into an internal form, the C++ code generator would produce a number of C++ declarations. The public portion of the main Foo class might look as follows:

```

class Foo : virtual public PhObject {
public:
    // Structors
    virtual ~Foo() {} ;// destructor

    // Foo methods
    virtual void bar() ;

    // Binding methods
    static Foo* bind(const PhOid& id) ;
    Foo* rebind(const PhOid& id) ;
};

```

Note that Foo will inherit some standard methods and data members (e.g., its object id) from the abstract Photon object class PhObject. The binding methods will be used by the binding template shown below. Since we want to hide the distinction between the use of Foo objects in the local address space and those accessed through proxies, the Foo class will only be an abstract base class, the real implementation of the class will be called MasterFoo and the proxy version of the class will be called ProxyFoo, both of which will inherit from the main Foo class:

```

class MasterFoo : public Foo, public PhMasterObject {
    // Structors
    MasterFoo() ; // default constructor
    MasterFoo(const PhOid& id) : PhObject(id) ; // init object id
    ~MasterFoo() ; // destructor

    // Operators
    void bar() { cout << "bar" << endl; };
};

class ProxyFoo : public Foo, public PhProxyObject {
    // Structors ProxyFoo() ;
    ProxyFoo(const PhOid &id) ;
    ~ProxyFoo() ;

    // Operators
    void bar() ; // Will invoke bar() on MasterFoo object remotely
};

```

Note that multiple inheritance is used so that each class can inherit a common interface from Foo and also inherit members specific to Master and Proxy objects respectively. Both classes have a constructor which sets the object id inherited from PhObject. The default constructor for PhObject could be made to either set the object id to a null value or to cause a new id to be generated when the object is constructed. Given this scheme a pointer to a Foo object could refer to either a MasterFoo or a ProxyFoo and allow them to be used interchangeably:

```

MasterFoo mfoo = id ; // create foo with specified object id
ProxyFoo pfoo = id ; // create proxy foo for given object id
Foo * mfoo_ptr = &mfoo,
    pfoo_ptr = &pfoo ; mfoo_ptr->bar() ; // invoke bar() method directly
    pfoo_ptr->bar() ; // invoke bar() method indirectly through proxy

```


Of course, this would still require the programmer to explicitly construct the proxy. This can be avoided using a C++ template class:

```
template<class obj>
class PhBind {
private:
    obj* _ptr ;

public:
    // Structors
    PhBind() : _ptr(0) {} ;
    PhBind(const PhOid &id) { bind(id); };
    PhBind(const PhBind<obj>& b) : _ptr(b._ptr) {} ;
    ~PhBind() {} ;

    // Operators
    obj* operator->() { return _ptr; };

    // Methods
    void bind(const PhOid &id) { _ptr = obj::bind(id); };
```

This template class hides the details of binding from the user through the use of a "smart" pointer. Note that it relies on the presence of the 'bind' method in the main Foo class to return an appropriately typed pointer. Although the bind method is part of the Foo class, the code generator will take care of the implementation, and the Photon class designer need not be troubled with it. Users of the template can use it exactly as they would use a pointer to a Foo object but don't have to be cognizant of the mechanisms used to bind it:

```
PhBind<Foo> foo(id) ; // Bind "foo" to object with given id
foo->bar() ; //invoke bar() method on object
```

The object referred to by 'id' might be a local MasterFoo object or a ProxyFoo object constructed on the fly and bound to a remote MasterFoo object through an appropriate communications channel, but the user need not be aware of this.

3.3.3 Naming

The ability to name an object is an important part of Photon. Although every Photon object has an object id, OIDs are system generated bit strings with no external meaning other than as a unique identifier for the object and thus convey little information to potential users of the object. Naming is a means of associating an object with an identifier which may be more meaningful to a human or organization.

Recall the dichotomy in the Unix(tm) file system between file names and inode numbers. The same dichotomy exists between Photon Object Names and Photon OIDs. The Photon Object Naming system is a catalog which given a name string (or other meaningful specifier) returns an object id. The OID may then be used to obtain access to the object.

The Photon Object Naming System, hereinafter called the catalog, is a system built strictly out of Photon objects. That is, the catalog is layered above most of the parts of Photon that implement Photon objects. The catalog is a loose hierarchy of Photon directory objects. The directory object

implements a lookup method. Given a directory object D, and a typed data item I (i.e. I is an instance of some Photon abstract type), P[I] (or P->lookup(I) if you prefer this notation) yields the OID of another Photon object. The type of the index argument I is by default a text string. However specialized directories may take other types of arguments to their lookup method.

Applying the lookup method to a directory object may return the OID of another directory object, thus enabling the construction of a hierarchical catalog. Photon does not require a single universal catalog. Starting with any directory as a "root", a hierarchical catalog will be seen. Just as it is expected that the catalog will be filled with meaningful entries by its users in order for it to be useful, it is expected that reasonable choices will be made for "root" directories. In particular, one Photon directory on each local system leads to a reasonable, standard organization for a catalog. An alternative might be one catalog "root" per workgroup, or per company, etc. Since at any point, a directory entry might point to another directory which might be remote to the "local" catalog, the hierarchical structure is merely a convenience.

The Photon directory object must of course implement additional methods to be generally useful. It should be possible for an application (at least some applications) to traverse a directory, to display all of the index entries. Traversal may not make sense in all cases (consider an Internet automounter implemented as a directory), but is generally useful.

One final point: users often think of owner id, access protection, size and date information as being part of the function of the directory, however, it need not be. In Unix(tm), this information comes from the inode, not from the directory. Similarly, in Photon, this sort of information would be part of the stable store that holds persistent objects (including files).

3.3.4 Location/Binding

Location/Binding is responsible for mapping a Photon object identifier into an object instance. Photon provides location independent invocation, like Cronus [WALKER90]. In Photon, however, the application builder has much more control over the location process than in Cronus. In Cronus, location is performed by the Cronus kernel in a fashion completely transparent to applications. Although it is often desirable for the system to take on all the burden of location, some applications might find it useful to be able to specify a location policy. Cronus has a hardwired policy of using the first object server found. Such a policy would result in a nearby PC being chosen over a distant supercomputer for a number crunching operation. Photon allows the application writer to replace the system's location policy with an application-specific one.

Photon provides support for application-specific location policies by placing the location function in a shared library rather than a separate ipc-kernel process. To customize location, the application writer just provides custom implementations of routines in the shared library that control location.

3.3.5 Photon ADT and Class Registration

Both Photon ADTs and classes will have descriptor objects that may be stored in individual files and used by the language veneer code generators to create appropriate interface and implementation code. For initial development and prototyping this scheme is sufficient since all of the files can be generated and used locally without needing to use anything more than the code generators, compilers, and linkers for the current environment. However, when an ADT package or Photon class is ready to be used in multiple applications or by multiple developers, it is highly desirable to have the means to publically register and distribute the package or class. Photon will provide this means in an ADT and class registration database.

The type registration database will provide a number of basic services related to the use of ADTs and classes:

- Object storage and retrieval

The principal service provided by the database, will be the ability to register ADT packages and Photon classes with server for subsequent retrieval. These will be registered by insertion of the descriptor object into the database. The database will also be able to manage source code generated for various language veneers and shared object code which implement ADT and class methods. The database will be distributed across the machines within a single domain.

- Name/id mapping

The database will support operations to translate between names and unique type identifiers. Given a class identifier, it will be able to provide the class name. Given a package name, it will be able to list the names and identifiers of ADTs contained in it. It will also be able to provide dependency information where appropriate; given a class name or identifier, it will be able to list parent and descendant classes and which ADT packages it requires.

- Name reservation

The database will allow programmers to reserve Photon class and ADT package and names and identifiers for future use, even before a definition is present. This will help developers avoid name clashes by allowing them to publish their intentions in advance.

Note that although the type registration database will have a well-defined interface, it will not necessarily be implemented in a single server process, as is the case with the Cronus Typedef Manager.

Section 3.4 Inter-Layer Modules

Two modules in the Photon architecture have functionality that is provided in every layer of the architecture, the Security and Monitoring modules.

3.4.1 Security

Photon will use an adaptation of the popular approach first described in [NEEDHAM78], which is based on message encryption and having a trusted authentication service. Other systems, for example [SATYAN89] and [BIRRELL85], were built without authentication and then had it fitted in later. Photon's basic initial architecture includes authentication, which should result in an overall cleaner design.

To lower the impact of network latency on establishing a communication path to a Photon object, the naming, location, binding, and authentication functions of Photon will be highly integrated. Other systems separate the binding and authentication function into separate servers. This means one round trip is needed to resolve the server's name, and another is required to obtain the necessary authentication information. By integrating these services, name resolution and authentication can be done in a single round trip. This approach to bundling naming and authentication together was also taken by [BIRRELL85].

3.4.2 Monitoring

Monitoring helps support the debugging and testing of distributed applications [JOYCE87], and can be used to conduct performance evaluation studies. Monitoring data can be used in real-time by adaptive algorithms to observe and adjust to a changing environment. Monitoring data can be used to automatically trigger administrative actions.

Many aspects of operation in each layer of the Photon architecture could be monitored. In the top layer, for example, usage patterns of Photon objects could be monitored. A record could be kept of what user invoked which method at what time. At the bottom layer, communications performance could be monitored. In the middle layer, contention in distributed shared memory could be monitored. [OSF DCE RFC 32] presents one version of what one might monitor in a distributed system. Activities in applications as well as within the Photon system itself could be monitored as well, to support testing and debugging. The monitoring data in this case can be application-specific.

The Photon architecture includes a Photon monitoring object that collects monitoring data, and replies to queries against its database of collected data. Modules report data concerning their activities to the Photon monitoring object. Applications can also report application-specific activities to the Photon monitoring object. Data is retrieved from the monitoring object by sending it a query. Applications that want to observe ongoing system activity can send the monitoring object a "live" query. The monitoring object then sends back any newly arriving data that satisfies the query until the application turns off the live query. A Photon sequence will be used to pass the monitoring data to the application. The elements of a sequence are filled in incrementally; the elements arrive at the receiver as they are filled in. The monitoring object can also support automatic administration and control activities. A query and a control action can be submitted to the monitoring object such that whenever the query is satisfied, the control action is executed.

Some adaptive applications might require the promptest possible reporting of monitoring data. For these applications, the latency of reporting monitoring data first to the monitoring object, and only

then to the application, might be unacceptable. Such applications will have the ability to request that modules directly send monitoring data to them, as well as the monitoring object. This approach trades off bandwidth for latency. The application receiving the monitoring data will have to examine and discard irrelevant monitoring data sent by the module. Having the sending module filter out monitoring data not of interest to the receiver might cause the monitored module to change its behavior when monitored significantly due to the increased computational load. Therefore some bandwidth will be wasted sending unneeded monitoring data, but this will be an increasingly worthwhile tradeoff as the cost of bandwidth decreases with respect to latency.

3.4.3 Administrative Domains

It is expected that Photon will not be used entirely within a single organization and that no one organization would be able to take responsibility for the administration of all Photon hosts. The configuration of Photon and user-provided services should be left to the local system administrator. To this end, Photon will group hosts into centrally administered "domains", such that every Photon host shall belong to exactly one domain. Hosts within the same Photon domain should be administered by the group or organization to which the domain is assigned, but need not actually be on the same local area network. Photon domains are roughly equivalent to "clusters" in Cronus.

Before describing domains in more detail, we should first speak a little about the Photon hosts of which they are comprised. A Photon host is a computer upon which Photon has been installed and which may support Photon client or server code. Every Photon host is identified by a unique host identifier which is assigned when Photon is installed. Photon will provide a service which will translate between Photon host ids and host names or system-specific host address (e.g., an Internet address), but for the purpose of communications within the Photon environment, Photon host ids will be sufficient. In Cronus, host identifiers or HOSTNUMs are implemented as Internet addresses and suffer from the resulting lack of generality: hosts with multiple addresses cannot be supported cleanly, changing the Internet address of a host invalidates existing HOSTNUMs, and there is no support for non-Internet addressing. Photon host ids, on the other hand, will have multiple internal representation types to support different addressing protocols, multi-ported hosts, and aliasing. Since some potential uses of host ids may require many bytes and others few and because we cannot anticipate in advance what the maximum space will be needed for all uses of host ids, Photon host ids will have variable length. Host ids will not contain the id of the domain to which the host belongs, but all Photon hosts will be able to identify their own domain.

Hosts within the same Photon domain will share common service providers for Photon services such as naming, authentication, and type registration. Clients which need to make use of such services for a remote domain will need to access the service provider for that domain. How related services in different domains communicate with each other will depend upon the service. For instance, authentication services in different domains will probably need to be explicitly introduced and authenticated to each other, but once that is done will be able to freely obtain authentication information from each other. The type registration service, on the other hand, is likely to require the explicit transfer of type information to and from remote domains.

Domains will be identified by unique identifiers which will be distributed along with the Photon installation files. Domain identifiers will be variable length in order to support further partitioning into subdomains for larger organizations. Photon will provide the means for users to obtain additional domain identifiers should they be needed.

Chapter 4. Resource Management in the Photon System

Photon is a distributed computing environment that supports high-performance computing. Photon has many features that allow an application to adapt over a wide range of system characteristics, such as global futures, shared memory, and proxy objects. Photon also accumulates and integrates a priori knowledge of an application's behavior in order to manage resources. Photon translates this knowledge into specifications for network resource allocation, such as a *flow specification*.

4.1 Introduction

A goal of Photon is to allow applications to adapt over a wide range of system characteristics. For example, a Photon application should run effectively on a shared memory multi-processor, or supercomputers connect together over a gigabit network. Photon should shield application users from resource management issues and help application developers to make adaptive systems.

Many specialized mechanisms have been invented to address resource management issues for systems with specific characteristics. A general solution to the resource management problem would be difficult to find, because even its sub-problems are hard optimization problems. Photon will address the general resource management problem by offering a framework for choosing specialized mechanisms based on a system's current characteristics. Hence, Photon must be aware of the system's characteristics and mechanisms that have been implemented to handle special cases.

Many systems are unaware of their operating environment; i.e. they make resource management decisions using no a priori knowledge. The claim is that collecting and maintaining knowledge about their operating environment is not worth the trouble. For example, in BSD Unix the paging algorithm dynamically determines the working-set size, which works fine for most programs. But for a program that inherently needs a large working set, this fact has to be rediscovered every time the program is run, and there is no mechanism for remembering this characteristic of the program.

On the other hand, large production systems tend to spend a considerable effort in measuring and analyzing system characteristics. The claim is that the system's size and relatively stable environment makes resource optimization profitable. For example, a whole industry supports "Capacity Management" of IBM mainframes. Some claim that a major reason why industry has not embraced client/server computing is the lack of support for system issues.

The characteristics of physical resources are also changing. Photon needs to handle the relatively high latency environment of future gigabit networks. But interfacing to these network will require detailed specification of traffic patterns, such as expected bandwidth and burst length. In order to use these resources effectively, Photon must translate an application's a priori knowledge of its usage patterns into precise communication requirements.

Photon must accumulate and integrate a priori knowledge of an application's system behavior. This knowledge is available at many different times, and from many different players, during the system development cycle. Photon offers hooks to capture this information at the appropriate time and place. Photon makes explicit the separation between the function of a distributed object and its realization on a set of networked resources.

This chapter introduces a terminology for analyzing the resource management process in Photon. Section 2 of this chapter introduces the general resource management process and show that several different types of information must be collected and analyzed. Section 3 of this chapter introduces the general system development process and shows that resource management information is available at different times. Section 4 of this chapter describes Photon's resource management process and shows the mechanism that Photon uses to collect, analyze and act upon resource management information.

4.2 A Resource Management Perspective of a System

A distributed application does not become a system until it is deployed on a set of networked computers. An application is coded to perform a specific function, but how well the application performs its function depends on how it consumes resources. For example, one would expect a program to run faster on a supercomputer than on a personal computer, but in some special cases this expected behavior may not be realized. Resource management is a framework in which to analyze system issues in isolation from functional issues.

Separating an application's functional properties from its system properties allows the distributed system to adapt to changes in resources or requirements. The object-oriented paradigm is a functional description of the application's data and processing, and does not describe system behavior. Photon uses resource management techniques to augment the object-oriented paradigm in order to handle system issues.

The same applications can have different system properties depending on how the resources are allocated to the application's functionality. *System properties* are requirements that resource management must meet. Examples of system properties are availability, performance, security, and cost. Resource management can change the *system characteristics* in order to fulfill desired system properties. Examples of system characteristics are resources, usage patterns, and resource allocation mechanisms.

Figure 4-1 shows the resource management view of system characteristics and properties. Resource management concentrates on system issues in isolation from the functionality of the application. System characteristics are the parameters that resource management can change, while system properties are the required behavior for the system. User traffic makes requests of physical resources in accordance with an resource allocation scheme. As a consequence of these system characteristics, the system has certain system properties. The resource consumption properties are concerned with the amount of resources consumed and their cost. The responsiveness properties are concerned with the quality of service the user received, such as availability, delay, throughput, and security. In the Photon context, these parameters are:

Usage Pattern: the workload on the system. Since Photon is an object-oriented system, workload is in terms of method invocations. The workload can be characterized by the time and frequency of method invocation among object clients and servers. These interactions can also be marked with their priority. Photon must be able to predict the expected workload which may include measuring past workload in order to predict the future.

Resources: physical components that have the capacity to do work. Photon must manage both the host resources and the network resources. The host resources include CPU, memory, disk, and specialized hardware, such as signal processing units. The network resources include managing connections between hosts. Photon must know which resources are available, the topology of how they are interconnected, and their current status.

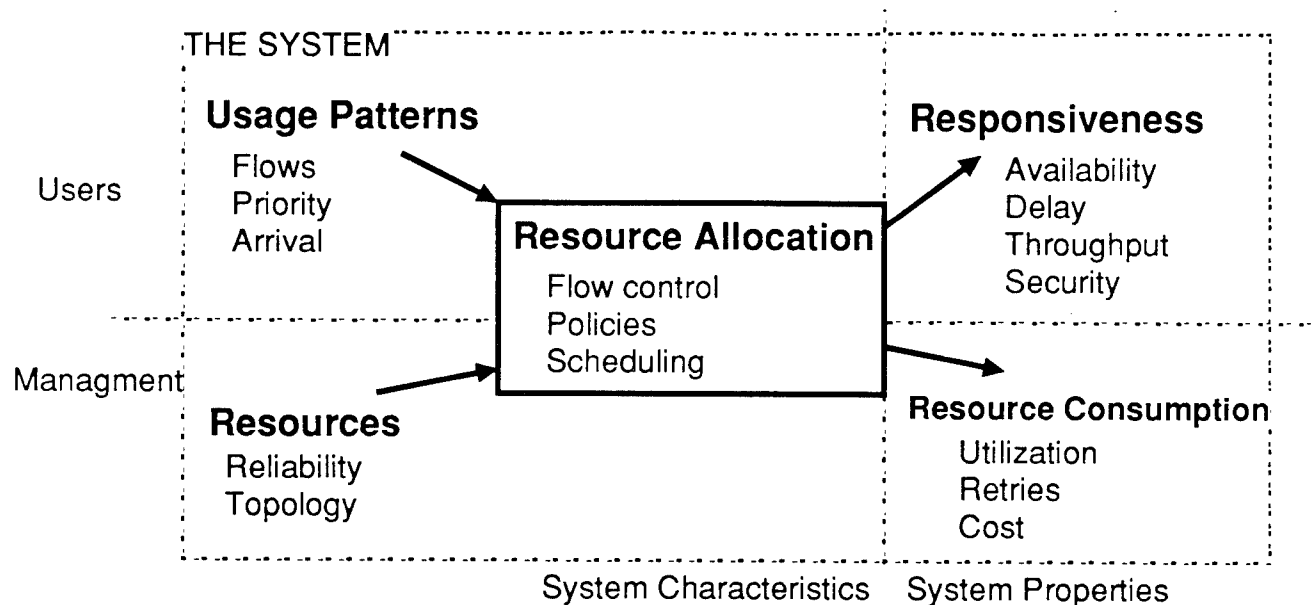


Figure 4-1: Resource management view of system characteristics and properties.

Resource Allocation: the algorithms used to assign work to workers. Photon supports run-time resource allocation. Photon has hooks for assigning resources when objects are created, connected, and used. Resource allocation algorithms can be fairly complicated and special purpose. Photon must have models of the expected system properties of an algorithm for a given usage pattern and resources.

Responsiveness: a catch-all term for how well a system performs its function. Traditionally, responsiveness has been expressed in terms of fault-tolerance, performance, timeliness, and security. Tom Lawrence has proposed that responsiveness can be reduce to a basis vector of Accuracy, Precision, and Timeliness [LAWRENCE]. Photon supports responsiveness at the level of objects. Photon accepts responsiveness requirements for interacting with objects, and verifies the actual responsiveness that it supplied.

Resource Consumption: measures the amount of resources a system consumes in order to perform its function. Photon keeps track of resource consumption for access control and billing reasons. User or applications might have rights to use only certain resources, and resources should not be overloaded, hence Photon must regulate the work flow.

To clarify these terms, here is an example. A mapping application expects to look up coordinates once a second; this is a usage pattern. The map object is stored on two servers, one locally and one remotely; this is a resource topology. The application uses the local server unless there is a failure; this is a resource allocation policy. The local server can handle two lookups a minute; this is a resource capacity. The application will use 50% of the server's capacity; this is its resource consumption. The applications has two sources for the map objects, so the interaction has high availability; this is responsiveness.

4.3 Overview of Resource Management Problems

The general problem of managing resources is intractable. The pragmatic solution is to break the general problem into a series of sub-problems. But even these sub-problems are typically hard optimization problems for which special case solutions are used. These sub-problem solutions usually assume that most of the system characteristics are fixed, and consider changes in only one variable. Thus, sub-problem solutions can be categorized into three groups depending on which system characteristic is considered variable (Figure 4-2).

The frequency with which system characteristics change determines the order in which subproblems should be solved. The slower changing characteristics can be handled with a longer time horizon than quickly changing characteristics. For example, in an office environment the number of workstations changes more slowly than the frequency applications are run on them. Hence there may be a yearly purchase of new office equipment whereas the application usage changes daily.

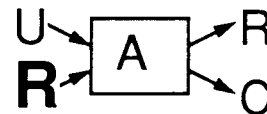
Solving a resource management problem depends on detailed knowledge and heuristics about the specific type of problem. Photon cannot supply one solution for all types of problems. Instead, Photon helps the system designer make specialized solutions for the key problems in their system. Photon provides several opportunities, or "times," when resource allocation can be performed. Photon mechanisms try to get the right information, to the right place, at the right time. Photon offers default resource management services, but they can be over-ridden by the object implementation. The following sections give example resource management problems and how Photon could support them.

4.3.1 Minimize Resources Problem

The *minimize resource* problem finds an inexpensive set of resources that meet responsiveness and

Minimize Resources

Fixed Usage Pattern and Allocation Mechanisms
Minimize Resources for Responsiveness Requirements
Examples: Network Design, Server Placement



Adaptive Allocation Mechanisms

Expected Usage and Expected Resources
Choose Allocation Mechanism to meet
Responsiveness Requirements
Examples: Policy Routing, Compression



Regulate Usage

Fixed Resources and Allocation Mechanisms
Adjust Usage to meet Utilization Requirements
Examples: Usage Scheduling, Flow Control

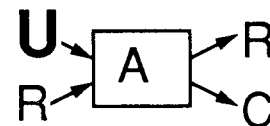


Figure 4-2: In order to make resource management problems tractable, most system characteristics are assumed constant and only one characteristic is changed.

utilization goals. The problem assumes fixed usage patterns and resource allocation mechanisms. The following are examples of the problem:

- **Network Design Problem:** An inexpensive network topology must be designed that can handle an expected traffic load while meeting responsiveness goals. The user traffic is defined in terms of transaction flow between sites and the responsiveness goals are in terms of transaction delay and throughput. The problem is to choose where to place switches and multiplexors, and how much bandwidth should be allocated between sites. Additional knowledge is necessary, such as the cost of switches, communication tariffs between sites, and how transactions map into communications load.
- **Server Placement Problem:** Replicated servers must be placed around the network such that they give adequate service, but do not consume very much network bandwidth. The location of the servers depend on the location of the users and how much traffic they generate. Also, backup servers must be defined which have enough capacity to still give adequate service in the case of failure.

Photon supports minimizing resources when objects are created, i.e. at *object instantiation time*. When an object is created, resources must be allocated to store the object's persistent state. The resources allocated depend on the object consumers' expected usage patterns and their responsiveness requirements. This problem is similar to the server placement problem.

Photon maintains expected usage patterns for objects and interfaces for specifying responsiveness requirements. In addition, Photon will measure the actual usage patterns and responsiveness to aid in determining future expectations.

4.3.2 Adaptive Allocation Mechanism Problem

This problem changes allocation mechanisms dynamically to compensate for a specific set of system characteristics. The new mechanism is customized for the expected usage pattern, available resources, and responsiveness requirements. The mechanisms take the form of "hard-coded" algorithms that are used on the critical path during run-time. Hence, the mechanisms tend to be efficient and simple, but tend to have only a small scope for which they are appropriate. The following are some examples of adaptive allocation mechanisms:

- **Policy Routing:** A path through the Internet must be chosen that meets a set of constraints imposed by the user. The current set of network resources are used and the constraints are both the expected usage pattern and the system properties. A policy-based route can be used until the system changes and then a new route must be constructed.
- **Compression:** A form of compression to be used between two hosts must be chosen from a set of possibilities. Resource constraints exist in the form of specialized hardware, network bandwidth, and CPU capacity. The content of the traffic also must be considered. For example, a text compression algorithm may actually expand binary speech data. The compression choice can be used until the system changes.

Photon supports adaptive allocation mechanisms when applications connect to objects, i.e. at *activation time*. When an application starts to use a distributed object, many decisions have to be made concerning which resources to use, and the run-time policies for interacting with the object. The choice of resources and policies to use depends on the expected usage pattern and the desired system properties.

Photon offers a special time, object activation time, for specifying requirements, and for setting up the resources and run-time policies. In addition, Photon will measure the actual usage and system properties to aid in determining future expectations.

4.3.3 Regulate Usage Problem

The *regulate usage* problem changes the workload characteristics so that resources will not be overloaded. The problem assumes fixed resources and allocation mechanisms, and must maintain a set of system properties. The following are examples:

- **Scheduling:** A task must be done by a certain time and each subtask demands specific resources. Scheduling determines an ordering for doing the subtasks that minimizes the time to complete the task. The problem is to resolve conflicts when subtasks need the same resources.
- **Flow Control:** Work is arriving faster than the resources can handle and there is no place to store waiting tasks. Flow control throttles the work generating process to a rate that matches the resource capacity.

Photon regulates workload when an application calls an object's method, i.e. at *invocation time*. Photon checks the state of the resources that implement the method, and uses a run-time policy to determine if the method should block. If the method blocks, the application can not generate any more traffic. Thus, the workload is regulated at its source.

Photon maintains information concerning the dynamic state of resources. This information will be collected as applications interact with resources. For example, network utilization can be determined by looking at transport-layer status.

4.4 Overview of the System Development Process

A *system development model* captures the evolution of a system over time, and the information flow between players. In the terminology of this chapter a system includes the application code, the resources used to run the application, and the people who maintain the application. This broad view is important, because systems are not static; they evolve over a long period of time.

As a system evolves, Photon captures general resource management information and continually refines it for specific contexts. This process of dynamic configuration makes Photon's applications both economical and powerful. A major goal of Photon is having the same application code run efficient on radically different sets of resources.

A taxonomy for resource management information explains *who* specifies *what*, and most importantly, *when*. As the system evolves, knowledge about system characteristics and requirements of system properties becomes more precise. For example, when the application is being implemented, the programmers have a vague understanding of how the system will be used. They make implicit optimizations based on these assumptions. Photon offers mechanisms for making these assumptions explicit and passing them on to later stages of system development. These earlier assumptions become the defaults for later decisions, but can be overridden.

Figure 4-3 is a time line of the evolution of objects, resources, and applications. The "times" are time horizons when decisions are made. Early on, very few decisions have been made and there is a wide latitude of possible choices. But in later times, many decisions have been made, and there are many constraints on remaining choices. Decisions also depend on constraints from other parts of

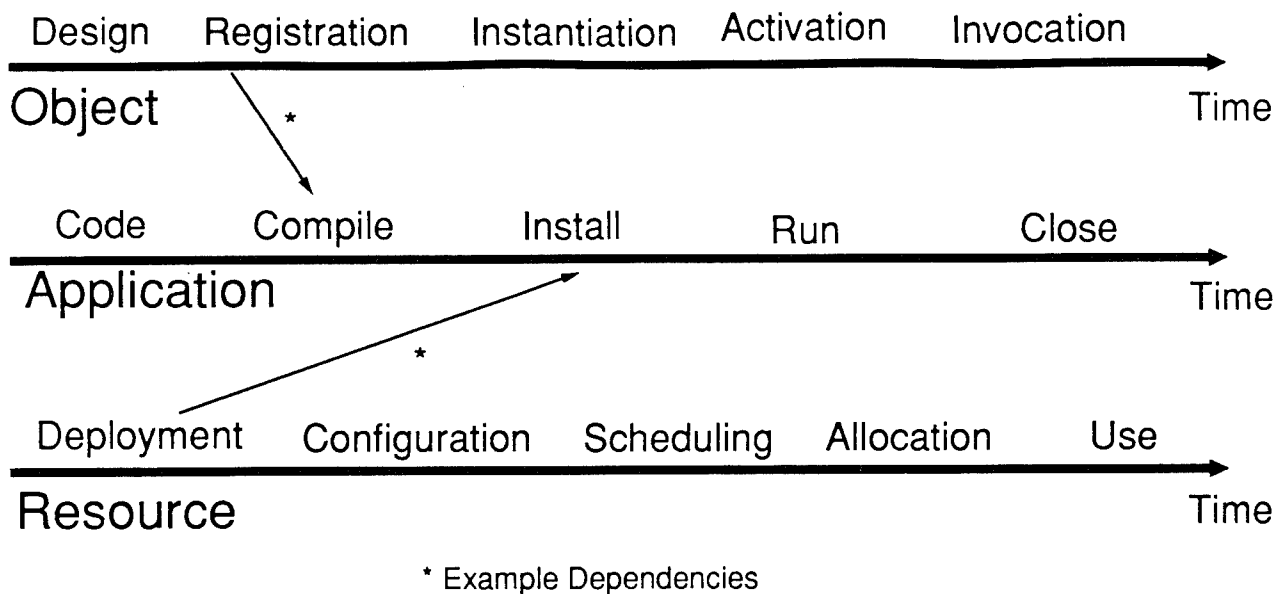


Figure 4-3: Evolution of objects, resources, and applications.

the system. For example, an application cannot be installed on a resource before the resource is deployed. Resource management information is available at different times, at different places, and from different people. The system development model captures the evolution of a system over time and the information flow among the players.

The rest of this chapter will concentrate on the run-time evolution of Photon objects: i.e. when objects are coded, created, and used. The following is a brief description of the important times in the evolution of objects.

- **Design Time** is when a programmer *designs* and *codes* an object class. The object developer defines the object interface using the Photon Interface Description Language (IDL). Besides specifying the functional interface to the object, the IDL can specify default resource management information. The object designer also defines the resource management granularity of the object, which involves partitioning the persistent state; defining invocation and activation policies; and coding alternative method implementations. Design time decisions have a broad scope; they effect all possible instances and all possible users.
- **Instantiation Time** is when an application *creates* an instance of the object. Resources are allocated to maintain the object's persistent state. The choice of which resources to use to maintain the state is based on the expected usage pattern of the object's consumers, and the desired system properties, such as replication for high availability. Photon's distributed shared memory is the principle mechanism for creating an object's persistent state. Instantiation time decisions begin to narrow in scope, as they pertain to a specific instance, but apply to all of its users.
- **Activation Time** is when an application *connects* to an object. Resources are allocated in anticipation of interactions between the application and the object. An application can have several activations of the same object, each with a different expected usage pattern and required system properties. Activation binds resources to functions. Activation sets up communications links and assigns implementations to methods. At activation time, Photon creates a proxy object in the application's address space which implements the chosen

invocation policies. Activation time decisions are limited to interactions between a specific application and a specific object.

- **Invocation Time** is when an application *calls* an object. Most of the resource management decisions have already been made, and the only task left is to regulate the workload, such as blocking when a remote host is overloaded, and recovering from anomalies in system characteristics, such as host failure. The proxy object will implement the invocation policy and use the resource bindings when its methods are called. Invocation time decisions are limited to a specific call to a specific object.

Notice that different players are involved at different times. Programmers design an object with only a vague idea of how objects will be used. System administrators create an object for their community of users. Finally, users connect to the object and actually do real work. As the time horizon shrinks, players have a more refined view about the actual system characteristics. Photon supports these views, by supplying hooks for collecting resource management information at these times, and for making resource allocation decisions.

Chapter 5. Summary of Technical Progress and Suggestions for Further Research

This chapter summarizes the technical progress made under the Integrated High Performance Distributed System (IHPDS) project, and makes suggestions about productive areas for future work and research.

5.1 Technical Progress

A very important accomplishment of the IHPDS project was refining the Photon architecture. This architecture is the first full-featured distributed computing environment tailored and optimized for high-performance environments. Although there are other systems that include a feature similar to one found in Photon, such as global futures, or distributed shared memory, Photon is the first architecture to offer anywhere near as complete a palette of solutions to the programmer of high-performance distributed application software.

Another important accomplishment of the IHPDS project was exploring resource management issues in depth, and incorporating hooks for resource management into the architecture. Effective dynamic resource management is critical for distributed applications to achieve truly high performance. The exact nature of the resources available to an application, and the contention for them from other applications, is generally not known at compile time; hence the importance of supporting dynamic resource management.

A third accomplishment of the IHPDS project was implementing the infrastructure to support the complete architecture. The infrastructure has been carefully designed and implemented; it is not "prototype" or "demo" software. We have built the high-performance infrastructure required at the base of the Photon architecture; it is optimized for use in high-performance environments.

Figure 5-1 indicates the portions of the Photon architecture that have been implemented. Shaded boxes represent the components that have been at least partially implemented. A description of the important C++ classes making up the implementation and the Photon Interface Definition Language supported can be found in the Integrated High Performance Distributed System User's Manual [BBN]. In the Basic Services Layer, both Abstract Data Types and Unique Ids have been fully implemented. A complete set of built-in Abstract Data Types is provided, and a language based on the CORBA IDL is provided along with a parser so that users can create their own new Abstract Data Types. In the Advanced Services Layer, Distributed Shared Memory and Object Implementation Support have both been partially implemented. A complete preemptable threads package was implemented for Object Implementation Support. A shared heap, along with synchronization primitives such as mutexes and condition variables, was implemented for Distributed Shared Memory. In the Object Services Layer, both the Interface Definition Language and Language Veneers were partially implemented. The portion of the Interface Definition Language and C++ Language Veneer that support Abstract Data Type definition was completely implemented.

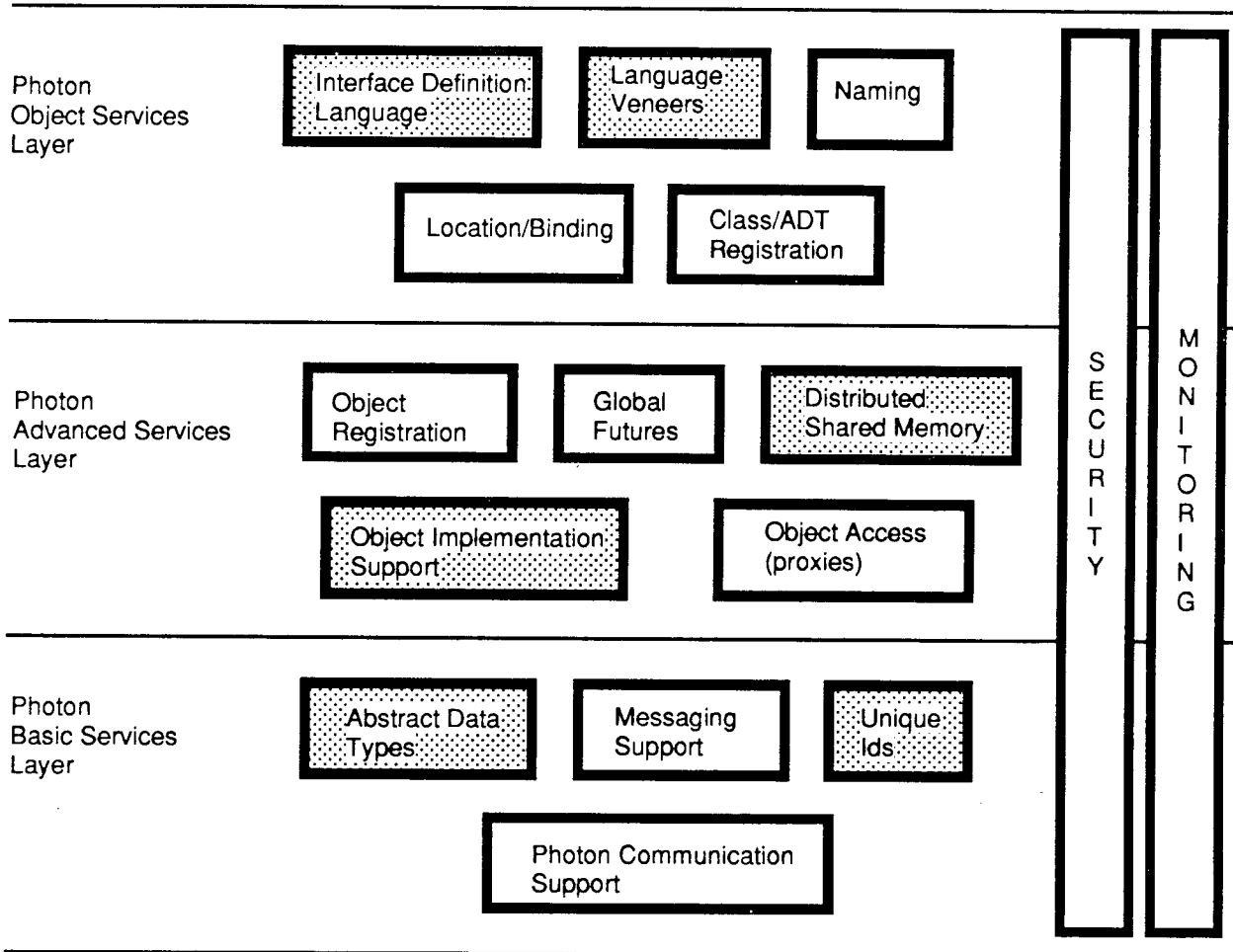
An important decision made concerning the Photon infrastructure was to conform to CORBA [CORBA93] whenever possible. Photon is a full-featured architecture; although it is optimized for high-performance environments, invariably it has a subset generic to general distributed environments. Although CORBA does not venture far into the world of high-performance, it does

have much to say about the interface of this subset of Photon. The decision was made to make this subset CORBA compliant. CORBA is a popular emerging standard; no other equivalent standard with its degree of acceptance currently exists. By being CORBA compliant, new Photon users that have had experience with other CORBA compliant systems will see many familiar features in Photon, and therefore only have to learn its novel features for high-performance. Another advantage of being CORBA compliant is that it offers the possibility of interoperating with objects built using other CORBA compliant systems.

While on the subject of CORBA, the difference of Photon from CORBA and other similar entities should be mentioned. Photon is different from such systems as Cronus, CORBA, DCE, and Libra because, unlike these systems, it is architected, implemented, and optimized solely for high performance environments.

In summary, the IHPDS project has developed a Photon architecture optimized for high performance environments, and produced a CORBA-compliant implementation of the base of that architecture.

Application Layer



Constituent Operating System Layer

Figure 5-1: Photon Implementation

5.2 Suggestions for Future Direction

The base of the Photon architecture has been implemented; an obvious and productive direction for future work is to continue implementing the rest of the architecture.

From largely anecdotal evidence, it seems to us that distributed shared memory has the potential for emerging as a very important and popular tool in distributed programming. Therefore expending additional effort in enhancing the Photon distributed memory architecture and implementing it fully would be productive. The implementation work on Photon concentrated on building a strong base for high-performance distributed shared memory.

Our work on resource management exposed us to the many exciting possibilities for resource management mechanisms that could be incorporated into Photon. More effort directed at designing and implementing such mechanisms would be productive. Other research efforts on resource management should be studied for concepts to incorporate into Photon and implement.

In summary, we suggest continuing the implementation of the full Photon architecture, and enhancing Photon's distributed shared memory and resource management architecture and implementation.

Chapter 6. References

- [BBN] "Integrated High Performance Distributed System User's Manual," Contract Number F30602-92-C-0192, CDRL 007, BBN Report Number 8004, July 1994.
- [BENNETT90] J. Bennett, J. Carter, and W. Zwaenepoel, "Munin: Distributed Shared Memory Based on Type-Specific Memory Coherence," Proc. 1990 Conference on Principles and Practice of Parallel Programming, ACM Press, New York, NY, 1990, pp. 168-176.
- [BIRRELL85] Andrew D. Birrell, "Secure Communication Using Remote Procedure Calls," ACM Transactions on Computer Systems (TOCS) 3,1, Feb. 1985, pp 1-14.
- [BERETS89] Berets, James C. and Richard M. Sands, "Introduction to Cronus," BBN Systems and Technologies Corporation, Technical Report 6986, January 1989.
- [C93] Helen Custer, "Inside Windows NT," Microsoft Press, Redmond Washington, 1993.
- [CLARK89] David D Clark, Van Jacobson, John Romkey, Howard Salwen, "An Analysis of TCP Processing Overhead," IEEE Communications Magazine, June 1989, pp. 23-29.
- [CLARK90] David D Clark and David L. Tennenhouse, "Architectural Considerations for a New Generation of Protocols," SIGCOMM 90.
- [COPLIEN] James O. Coplien, "Advanced C++: Programming Styles and Idioms," Addison-Wesley, Reading, Mass., 1992.
- [CORBA93] Digital Equipment Corporation, Hewlett-Packard Company, HyperDesk Corporation, NCR Corporation, Object Design, Inc., SunSoft, Inc., "The Common Object Request Broker: Architecture and Specification", OMG.
- [DRUSCHEL93] Peter Druschel, Mark Abbott, Michael Pagels, Larry Peterson, "Network Subsystem Design," IEEE Network, Vol. 7 No. 4, July 1993, pp. 8-17.
- [GRIMSHAW92] Andrew Grimshaw, "Easy-to-Use Object-Oriented Parallel-Processing with Mentat," Technical Report No. Cs-92-32. University of Virginia.
- [HALSTEAD85] Halstead, R., "Multilisp: A Language for Concurrent Symbolic Computation," ACM Transactions on Programming Languages and Systems, vol. 7, no. 4, October 1985.
- [JOYCE87] Jeffrey Joyce, Greg Lomow, Konrad Slind, Brian Unger "Monitoring Distributed Systems," ACM Transactions on Computer Systems, Vol. 5, No. 2, May 1987, pp. 121-150.
- [KRST93] M. Frans Kaashoek, Robbert Van Renesse, Hans Van Staveren, and Andrew S.

Tanenbaum, "FLIP: An Internetwork Protocol for Supporting Distributed Systems," ACM TOCS, Vol. 11, No. 1, Feb. 1993, p. 73 ff.

[LAWRENCE] Lawrence, Thomas, personal communication.

[LISKOV85] Liskov, Barbara, "The Argus Language and System," Lecture Notes in Computer Science 190, pp. 343-430, Springer-Verlag, 1985.

[LISKOV88] Barbara Liskov, T. Bloom, D. Gifford, R. Scheifler, and W. Weihl, "Communication in the Mercury System," Proc. of the 21st Annual Hawaii International conference on System Sciences, January 1988, pp. 178-187.

[NEEDHAM78] Roger M. Needham, and Michael D. Schroeder, "Using Encryption for Authentication in Large Networks of Computers," CACM 21,12, Dec. 1978, pp. 993-998.

[OSF DCE RFC 32] R. Friedrich, "Requirements for Performance Instrumentation of DCE RPC and CDS services."

[POSIX 1992] POSIX P1003.4a, "Threads Extension for Portable Operating Systems", IEEE.

[RASHID86] R. F. Rashid, "Threads of a New System," Unix Review, Vol. 4., No. 8, Aug. 1986, pp. 37-49.

[RINARD93] Martin C. Rinard, Daniel J. Scales, and Monica S. Lam, "Jade: A High-level Machine-Independent Language for Parallel Programming," IEEE Computer, June 1993, pp. 28-38.

[SATYAN89] M. Satyanarayan, "Integrating Security in a Large Distributed System," ACM TOCS 7,3, Aug. 1989, pp. 247-280.

[SCHANTZ85] Schantz, Richard E. and Robert H. Thomas, "Cronus, A Distributed Operating System: Functional Definition and System Concept," BBN Laboratories, Technical Report 5879, June 1982, Revised January 1985.

[SHIRLEY92] John Shirley, "Guide to Writing DCE Application," O'Reilly and Associates, Inc., 1992.

[STEINER88] Jennifer Steiner, Cliff Neuman, and Jeff Schiller, "Kerberos: An Authentication Service for Open Network Systems," Winter USENIX 1988.

[THEKKATH93] Chandramohan A. Thekkath and Henry M. Levy "Limits to Low-Latency Communication on High-Speed Networks, ACM Transactions on Computer Systems, May 1993, pp 179-203.

[TOUCH93] Joseph D. Touch, "Parallel Communication," INFOCOMM 93.

[WALKER90] Walker, Edward; Richard Floyd, Paul Neves, "Asynchronous Remote Execution in Distributed Systems," Proceedings of the 10th International Conference on Distributed Computer systems, IEEE Computer Society, May 1990.

[WILSON92] Linda F. Wilson, Mario J. Gonzalez, and Michael W. Cruess, "Experiences in High Performance Computing with Pleiades and ESP," Proceedings of the First International

Symposium on High-Performance Distributed Computing, IEEE Computer Society, 1992, pp. 67-76.

[ZHOU90] Songnian Zhou, Michael Stumm, and Tim McInerney, "Extending Distributed Shared Memory to Heterogeneous Environments," Proceedings of the 10th International Conference on Distributed Computing Systems, IEEE, 1990, pp. 30-37.

MISSION
OF
ROME LABORATORY

Mission. The mission of Rome Laboratory is to advance the science and technologies of command, control, communications and intelligence and to transition them into systems to meet customer needs. To achieve this, Rome Lab:

- a. Conducts vigorous research, development and test programs in all applicable technologies;
- b. Transitions technology to current and future systems to improve operational capability, readiness, and supportability;
- c. Provides a full range of technical support to Air Force Materiel Command product centers and other Air Force organizations;
- d. Promotes transfer of technology to the private sector;
- e. Maintains leading edge technological expertise in the areas of surveillance, communications, command and control, intelligence, reliability science, electro-magnetic technology, photonics, signal processing, and computational science.

The thrust areas of technical competence include: Surveillance, Communications, Command and Control, Intelligence, Signal Processing, Computer Science and Technology, Electromagnetic Technology, Photonics and Reliability Sciences.